Titre: Title:	De nouvelles perspectives d'utilisation des logs dans un contexte de sécurité informatique
Auteur: Author:	Edwin Bourget
Date:	2016
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bourget, E. (2016). De nouvelles perspectives d'utilisation des logs dans un contexte de sécurité informatique [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2233/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2233/
Directeurs de recherche: Advisors:	Alejandro Quintero
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

DE NOUVELLES PERSPECTIVES D'UTILISATION DES LOGS DANS UN CONTEXTE DE SÉCURITÉ INFORMATIQUE

EDWIN BOURGET
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DE NOUVELLES PERSPECTIVES D'UTILISATION DES LOGS DANS UN CONTEXTE DE SÉCURITÉ INFORMATIQUE

présenté par : BOURGET Edwin
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doctorat, présidente
M. QUINTERO Alejandro, Doctorat, membre et directeur de recherche
Mme BELLAÏCHE Martine, Ph. D., membre

DÉDICACE

à mes parents Daniel et Martine Bourget

REMERCIEMENTS

Je remercie Alejandro Quintero pour ses encouragements, sa disponibilité, ses conseils et son aide efficace.

Je remercie l'École Polytechnique de Montréal pour m'avoir permis d'étudier au sein d'une prestigieuse université québécoise.

Je remercie Telecom SudParis de m'avoir offert la chance de participer à cet échange.

Et je remercie ma famille pour son soutien à toute épreuve.

RÉSUMÉ

L'explosion des connexions des systèmes industriels liées au cyberespace a engendré une augmentation conséquente des attaques informatiques. Afin de garantir la sécurité des systèmes industriels il est devenu vital de développer de nouveaux outils de surveillance. La mise en place de tels dispositifs peut cependant avoir des conséquences sur l'outil de production, causant des ralentissements ou s'avérant handicapant pour le fonctionnement. Dans un tel contexte, mettre au point des méthodes non intrusives qui assurent leur sûreté est un enjeu majeur pour la sécurité des systèmes industriels.

Les logs sont des ensembles séquentiels de messages produits par un programme dont le rôle est de conserver un historique de l'exécution. Leur génération et leur consultation n'interfèrent pas avec le programme dont ils sont issus, si ce n'est en consommant des ressources du système d'exploitation hôte. Ce caractère les rend particulièrement précieux afin de respecter la contrainte de non-intrusion sur les systèmes industriels que doivent respecter les outils de surveillance.

La détection d'anomalies au travers des logs est un problème qui a déjà été couvert dans la littérature, en utilisant une grande variété de modèles. Si l'on considère qu'une attaque est une utilisation anormale du système laissant des traces différentes dans les fichiers de logs, la transposition de ces méthodes constitue une première approche intéressante du problème ainsi qu'un excellent point de départ pour apporter une solution.

En suivant cette démarche, nous proposons deux modèles permettant de détecter des attaques en exploitant des données présentes dans les logs : un automate fini et un réseau de neurones. Ces deux modèles reçoivent des données différentes, extraites des logs. Précisons que le système étudié est un serveur Web, générant donc des logs Web et que les attaques qu'il subit sont parmi les plus répandues.

L'automate travaille sur les requêtes traitées par le serveur et cherche à reconstituer le parcours du client sur le site Web. L'hypothèse est que si l'automate a été construit en reprenant toutes les utilisations normales possibles du site Web, une trace non reconnue par l'automate ne correspondra donc pas à un comportement normal du client et sera donc labellisée comme résultant d'une attaque. Ses paramètres d'entrée ont été adaptés à deux attaques particulières bien que les tests aient été effectués sur quatre attaques différentes.

Le réseau de neurones choisi est un perceptron multicouches dont les paramètres d'entrée sont des vecteurs résumant des traces d'exécution. Le rôle du réseau de neurones sera, étant donné

le résumé d'une trace, de déterminer si celle-ci correspond à un comportement normal ou à une attaque et, dans le cas d'une attaque, de préciser laquelle. Le réseau de neurones a également été pensé pour détecter les deux mêmes attaques que l'automate mais ses paramètres d'entrée ont été adaptés pour pouvoir répondre à un cas plus général.

Concernant les résultats des expériences destinées à valider les capacités des deux modèles, l'automate a parfaitement répondu aux attentes. Il s'est avéré capable de détecter avec certitude deux des types d'attaques pour lesquelles il a été pensé, mais n'a pas été en mesure d'identifier les deux autres de nature trop différentes. Les traces correspondant à des comportements normaux ont été également correctement reconnues par l'automate. Le réseau de neurones a eu, quant à lui, des résultats plus frustrants, avec un pourcentage de réussite aux environs de 30%. Une version simplifiée du modèle a permis d'augmenter ce pourcentage à plus ou moins 60%.

Le réseau de neurones a eu de moins bons résultats pour différentes raisons. Notamment, avec du recul, on peut considérer que le modèle lui-même n'était pas adapté au problème, en particulier ces paramètres qui ne permettaient pas de croiser les logs du serveur Web avec d'autres (comme ceux générés par le système d'exploitation) afin d'en extraire de nouveaux comportements. La raison principale reste cependant que le modèle devait détecter des attaques trop différentes. Contrairement à l'automate qui était dédié au déni de service et à l'attaque par force brute, le perceptron a cherché, dès sa conception, à détecter les quatre attaques. C'est pourquoi il a perdu en efficacité sur les deux attaques principales.

Pour autant, le travail de recherche reste positif. L'objectif de ce projet était avant tout d'étudier des techniques de détection d'intrusion et de montrer le potentiel de l'utilisation de logs autour desquels construire des modèles de surveillance de systèmes industriels. Ce but a été atteint et la perspective de mettre au point de tels dispositifs peut être envisagée.

ABSTRACT

A sizable increase in the number of computer attacks was due to the outburst of connections between industrial systems and the cyberspace. In order to ensure the security of those systems, it became vital to develop new monitoring solutions. Yet, setting up those mechanisms may have consequences on the production tool, leading to falloffs or crippling its normal functioning. In this context, developing non-intrusive methods that ensure reliability in a major concern in the security of industrial systems.

Logs are sequential sets of messages produced by a program whose role is to keep track of a historic of execution. Their generation and their reading do not interfere with the program that creates them, besides consuming resources of the host operating system. This feature makes them very valuable when trying to respect the non-intrusion constraint that monitoring tools must abide by when dealing with industrial systems.

Anomaly detection through logs is a problem that was studied in the literature, using several models. If one considers an attack as an abnormal use of a system, writing different traces in the log files, the translation of these methods establish an interesting first approach of the problem as well as an excellent starting point to bring a solution.

Following this process, we propose two models allowing to detect attacks by using data contained in logs: a finite state automaton and a neural network. There two models receive different data extracted from logs. Let us point out that the system studied is a web server, generating web logs and that the attacks it underwent are amongst the most spread ones.

The automaton works on request handled by the server and seeks to rebuild the journey of the client on the website. The hypothesis is this: if the automaton was built using all the possible paths a normal client can follow, a non-recognized trace would not correspond to a normal behavior and thus will be labeled as the result of an attack.

The neural network chosen is a multilayer perceptron which input parameters are vectors summarizing execution traces. The role of the network will be, given the summary of a trace, to determine if it corresponds to a normal behavior or an attack, and in the case of an attack, which one.

Regarding the results of the experiences whose objective was to validate the capacities of the two models, the automaton addressed our needs. It proved itself able to detect two types of attacks with certainty, but was incapable of identify two others. Traces corresponding to a normal behavior have also been recognized by the automaton. The neural network, as for it,

had much more frustrating results, with an success rate of about 30%. A simplified version of the model was able to increase this rate to about 60%.

The neural network may have had lower results for various reasons. With more hindsight, the model itself was not the most suited for the problem. Parameters may not have been convenient for the model, et would not allow any cross analysis with others (such as logs generated by the host operating system) in order to extract new behaviors either. Though, the main reason is that the model had to detect attacks that were too much different. Contrary to the automaton that was dedicated to the denial of service and the bruteforce attack, the perceptron tried, according to its design, to detect the four attacks. This is why it lost its efficiency on the two mains attacks.

Nevertheless, the overall work remains positive. The objective of this project was to study intrusion detection methods and to demonstrate the potential of the use of logs in building models to monitor industrial systems. This goal was reached and the perspective to develop such tools can be considered.

TABLE DES MATIÈRES

DÉDICACE iv

REMERCIEMENTS v

RÉSUMÉ vi

ABSTRACT viii

TABLE DES MATIÈRES x

LISTE DES TABLEAUX xiii

LISTE DES FIGURES xiv

LISTE DES SIGLES ET ABRÉVIATIONS xv

LISTE DES ANNEXES xvi

CHAPITRE 1 INTRODUCTION 1
- 1.1 Définitions et concepts de base 1
- 1.2 Éléments de la problématique 3
- 1.3 Objectifs de recherche 4
- 1.4 Plan du mémoire 4

CHAPITRE 2 REVUE DE LITTÉRATURE 6
- 2.1 Les logs 6
  - 2.1.1 Les logs du serveur Apache 6
  - 2.1.2 L'écriture des logs 7
- 2.2 Les automates 8
  - 2.2.1 Quelques définitions 8
  - 2.2.2 Construire un automate reconnaissant une exécution : kTail 9
  - 2.2.3 Une amélioration de kTail : kBehavior 10
- 2.3 Les réseaux de neurones 11
  - 2.3.1 Généralités sur les réseaux de neurones 11
  - 2.3.2 Apprentissage 12
  - 2.3.3 Apprentissage supervisé ou non ? 13

2.3.4 Competitive Learning 13
2.3.5 Des machines à vecteur de support 14
2.3.6 Des réseaux de neurones pour faire de l'agrégation 14
2.4 D'autres techniques d'analyse de log 16
2.4.1 Utilisation d'arbres de décision 16
2.4.2 Un classifieur bayésien naïf 18

CHAPITRE 3 LE MODÈLE 19
3.1 Deux modèles très différents mais pas incompatibles 19
3.2 Identifier les traces et extraire des données 20
3.2.1 Le cas de l'automate 21
3.2.2 Le cas du perceptron 22
3.3 Un automate éduqué pour reconnaître ce qu'il a déjà vu 22
3.3.1 L'apprentissage 22
3.3.2 kBehavior 23
3.3.3 La détection d'attaques 30
3.4 Inférer une fonction de décision 30
3.4.1 Exploiter les données 30
3.4.2 Des paramètres supplémentaires 33
3.4.3 La sortie 34
3.4.4 La topologie du réseau 34
3.4.5 L'apprentissage 35
3.5 L'utilisation du réseau 37

CHAPITRE 4 EXPÉRIMENTATIONS ET RÉSULTATS 38
4.1 D'où proviennent les données utilisées ? 38
4.2 Le serveur Apache 38
4.3 Les attaques 40
4.3.1 Déni de service 40
4.3.2 Bruteforce 42
4.3.3 Injection de code 42
4.3.4 Cross-Site Scripting 43
4.4 Le protocole expérimental 44
4.4.1 Générer des données 45
4.4.2 Raffiner les logs 45
4.4.3 L'apprentissage des modèles 47
4.5 Les résultats 48

4.5.1 L'automate 49
4.5.2 Le réseau de neurones 51

CHAPITRE 5 CONCLUSION 57
5.1 Synthèse des travaux 57
5.2 Les limites de la solution proposée 58
5.3 Les améliorations futures 58

RÉFÉRENCES 60

ANNEXES 62

LISTE DES TABLEAUX

Tableau 1.1 Un exemple de logs 2
Tableau 3.1 Des traces d'exécution 21
Tableau 3.2 Cinq traces identifiées 22
Tableau 4.1 Des entrées pour l'automate 46
Tableau 4.2 Des entrées pour le réseau de neurones 47
Tableau 4.3 Résultats de la reconnaissance des traces pas l'automate 50
Tableau 4.4 Résultats des tests sur le perceptron 52
Tableau 4.5 Résultats des tests sur les modèles simplifiés 54

LISTE DES FIGURES

Figure 2.1 Un exemple d'automate 8
Figure 2.2 Un exemple de perceptron multi-couches 12
Figure 2.3 L'hyperplan d'équation $y = x$ sépare les deux classes "plus" et moins" 14
Figure 2.4 Topologie d'un réseau de neurones en grille 15
Figure 2.5 Un arbre de décision 17
Figure 3.1 La génération du premier automate 27
Figure 3.2 L'automate à fusionner 27
Figure 3.3 Résultat de `kBehavior` sur la première trace 27
Figure 3.4 L'automate à fusionner 28
Figure 3.5 Résultat de `kBehavior` sur la seconde trace 28
Figure 3.6 L'automate à fusionner 28
Figure 3.7 Résultat de `kBehavior` sur la troisième trace 29
Figure 3.8 Résultat de `kBehavior` sur l'ensemble des traces 29
Figure 3.9 La sigmoïde 35
Figure 4.1 Le site web 39
Figure 4.2 Résultat de `kBehavior` sur l'ensemble des traces 50
Figure 4.3 Attaque par force brute 56
Figure 4.4 Déni de service 56
Figure 4.5 Injection SQL 56
Figure 4.6 Attaque XSS 56
Figure A.1 Le logo de Microsoft 62
Figure A.2 [2 - 10 - 3] 63
Figure A.3 [2 - 10 - 10 - 3] 63
Figure A.4 [2 - 50 - 50 - 3] 63
Figure A.5 [2 - 50 - 50 - 50 - 50 - 3] 63

LISTE DES SIGLES ET ABRÉVIATIONS

IP	Internet Protocol
RFC	Request For Comments
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
SVM	Support Vector Machine
DoS	Denial of Service
SOM	Self-Organizing Map
ART	Adaptive Resonance Theory
NAT	Network Address Translation
COTS	Commercial of the Shelf
SQL	Structured Query Language
XSS	Cross-Site Scripting
PPI	Page Popularity Index
IIS	Internet Information Services
HTML	Hypertext Markup Language
OWASP	Open Web Application Security Project
LDAP	Lightweight Directory Access Protocol

LISTE DES ANNEXES

Annexe A Influence des couches cachées 62
Annexe B Un peu de code 64

CHAPITRE 1 INTRODUCTION

Il n'existe pas de système parfaitement sécurisé. Même si vous débranchez votre ordinateur et l'entreposez dans un hangar, il y aura toujours une personne capable de le rebrancher. Les vulnérabilités sont donc inhérentes aux systèmes informatiques. Mais, s'il n'est pas possible de toutes les supprimer, on peut néanmoins les surveiller. En parlant de surveillance ici, nous faisons référence à deux réalités. La première est l'analyse en temps réel de l'utilisation qui est faite du système : cette opération permet de détecter une attaque, voire de l'empêcher, et d'alerter les administrateurs en cas de problème. La seconde est l'étude des traces laissées par l'exploitation d'une vulnérabilité : cela peut permettre de comprendre la méthode employée par l'attaquant, de mettre en place des contre-mesures ou même de mieux sécuriser l'application.

La sécurité informatique a un coût. On pense spontanément aux experts en sécurité qu'il incombe de payer pour vérifier et sécuriser son système ou bien aux produits tels que les antivirus que l'on doit acheter afin de se protéger contre des programmes malveillants. C'est une dépense qu'il est aisé de budgétiser et que les entreprises sont enclines à payer. Cependant, les opérations de chiffrement, d'analyse ou de test consomment des ressources : du temps et de la puissance de calcul. Dans un tel contexte et afin qu'une solution de sécurité soit pertinente, il est nécessaire que celle-ci ait le moins d'impact possible sur le système déjà existant.

1.1 Définitions et concepts de base

On utilise les fichiers de logs pour trouver des erreurs dans les programmes. Pourquoi ne pas les utiliser pour trouver des attaques ?

Les fichiers de logs sont des ensembles séquentiels de messages émis par un programme informatique rendant compte de son exécution (voir tableau 1.1). Un log est un message texte, avec des métadonnées ou non, contenant des informations sur un évènement s'étant produit au sein du programme. L'ensemble des logs révèle donc l'histoire du programme. Ce caractère historique est très précieux quand il s'agit d'identifier un problème qui est survenu. Il suffit d'analyser le fichier de logs pour diagnostiquer le problème et prendre les mesures nécessaires. Par ailleurs, les logs sont générés en temps réel. Il est donc possible de s'en servir pour surveiller un programme pendant son exécution.

Tableau 1.1 Un exemple de logs

192.168.100.1 - - [10/Mar/2016 :15 :13 :30 -0800] "GET /index.html HTTP/1.1" 200 632 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :30 -0800] "GET /11.html HTTP/1.1" 200 341 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :31 -0800] "GET /12.html HTTP/1.1" 200 341 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :31 -0800] "GET /13.html HTTP/1.1" 200 347 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :31 -0800] "GET /index.html HTTP/1.1" 200 576 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :32 -0800] "GET /21.html HTTP/1.1" 200 372 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :32 -0800] "GET /231.html HTTP/1.1" 200 341 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :32 -0800] "GET /21.html HTTP/1.1" 200 372 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :32 -0800] "GET /221.html HTTP/1.1" 200 343 "-" "-"
192.168.100.1 - - [10/Mar/2016 :15 :13 :33 -0800] "GET /222.html HTTP/1.1" 200 347 "-" "-"

La génération d'un fichier de logs fait partie de l'exécution normale d'un programme. Les utiliser ne nécessite donc pas de modification du système existant. En réalité, il est possible de désactiver la génération des logs mais cette opération est fortement déconseillée, car il devient par la suite quasiment impossible de retracer un bug dans le programme. Par conséquent, le postulat selon lequel le système informatique que l'on désire sécuriser génère des logs est une hypothèse raisonnable, vérifiée dans l'immense majorité des cas.

Une caractéristique quasi inhérente aux fichiers de logs est leur volume. La quantité de logs générés par un système industriel peut aisément représenter plusieurs centaines de gigaoctets par jour. Des outils d'analyse de logs pour entreprise, tels que Splunk, peuvent traiter jusqu'à dix téraoctets de données par jour. Dans un tel contexte, il est impératif de concevoir des algorithmes d'analyse ayant la complexité la plus basse possible.

Actuellement, les logs sont principalement utilisés pour retracer des erreurs. Quand un programme rencontre une erreur trop importante, il plante. Lorsque cela arrive, toute la mémoire du programme est perdue, ainsi que toute possibilité, pour la personne chargée de corriger le problème, de savoir dans quel état il se trouvait. La seule information résiduelle se trouve dans les fichiers de logs, qui constituent alors une source d'information précieuse.

Étant donné la quantité potentiellement énorme de logs à analyser en cas de problème, de nombreuses techniques automatisant ces processus ont vu le jour. Ces techniques considèrent des logs quelconques et les traitent donc comme des chaînes de caractères sans métadonnées ((8)) dont il est nécessaire d'extraire de l'information ((19)). La plupart des méthodes employées sont empruntées au domaine de l'apprentissage machine avec, par exemple, des arbres de décisions ((13)), des classifieurs Bayésiens naïfs ((14)), de l'analyse en composantes principales ((20)) ou des automates à nombre fini d'états ((12)). On remarque ici la grande variété des origines de ces techniques, qu'elles représentent une approche probabiliste, statistique ou déterministe du problème.

Il faut cependant noter qu'il est possible de regrouper les logs en différentes catégories. On trouve par exemple les logs réseaux qui servent à la surveillance et à l'analyse de la performance du réseau qui les génère. Il existe aussi des logs de sécurité qui sont gérés par les systèmes d'exploitation uniquement et qui enregistrent des informations relatives à sa sécurité. Les logs web sont générés par les serveurs web et ont des formats spécifiques et normalisés. Ces derniers sont, en particulier, les logs les plus analysés ; leur utilisation allant des attaques aux motifs d'accès ou de trafic ((8)). Enfin, les logs générés par la plupart des applications ne font partie d'aucune catégorie spécifique. En effet, il est difficile de les harmoniser autour de caractéristiques communes. Néanmoins, ils peuvent être universellement utilisés pour de la détection d'anomalies.

1.2 Éléments de la problématique

En anglais, "to hack" veut dire "bidouiller", c'est-à-dire détourner un objet ou un programme de sa fonction initiale. Même si ce terme est désormais galvaudé et représente beaucoup d'autres choses, il n'est pas rare que pour "hacker" un système, il faille en faire une utilisation non-conventionnelle. Une telle utilisation laisse des traces différentes de celles du fonctionnement normal du programme. Ces différences sont de la même nature que celles des erreurs d'exécution que les algorithmes présentés précédemment ont pour objectif de traquer. Bien qu'il semble alors naturel d'utiliser des logs pour détecter une attaque, cet emploi aussi direct des logs n'est que peu couvert dans la littérature.

Il devient alors intéressant de traiter le problème de la détection d'intrusion sous un angle nouveau, en transposant, dans un contexte de sécurité, des méthodes servant à trouver des erreurs d'exécution. Évidemment, toutes les techniques ne sont pas transposables avec la même efficacité, car elles sont induites par des spécificités du problème ou des éléments recherchés qui ne se retrouvent pas nécessairement dans la détection d'attaque. Il ne s'agit donc pas de comparer toutes les méthodes existantes afin de trouver laquelle est la plus performante sur un problème donné mais de considérer quelques attaques différentes, et de chercher à les détecter avec un algorithme servant à trouver des anomalies d'exécution, ou une adaptation d'un tel algorithme, et de motiver un tel choix.

Les méthodes de détection d'anomalies ne sont pas toujours conçues pour analyser le système en permanence. Par conséquent, certaines ont des coûts algorithmiques pouvant être prohibitifs. Un système détectant des intrusions une fois que l'attaque est terminée présente donc un intérêt limité dans notre situation. Attention, cela peut parfois être extrêmement précieux car analyser une attaque permet de comprendre comment celle-ci a été réalisée et comment s'en prémunir par la suite. Cependant, dans le cas qui nous intéresse, il s'agira avant tout de

construire un modèle capable d'alerter un administrateur qu'une attaque est en cours. Les coûts algorithmiques devront donc permettre une analyse des logs en temps réel, dans l'idéal en coût linéaire.

1.3 Objectifs de recherche

L'objectif de ce projet est de transposer des techniques de détection d'anomalies au sein d'un programme dans un contexte de sécurité afin de reconnaître des attaques informatiques. Ces modèles devront s'inscrire au sein d'un protocole expérimental automatisé. Gardons à l'esprit qu'il est très difficile, voir impossible, de développer une méthode universelle capable d'analyser toutes les attaques. Nous allons donc chercher à détecter deux attaques courantes en utilisant des modèles qui nous semblent adaptés. Nous allons également tester nos modèles sur deux des vulnérabilités les plus répandues.

La démarche proposée afin d'atteindre de tels objectifs est en quatre étapes :

1. Le développement de robots générant du trafic légitime et fallacieux sur un site web ;
2. L'extraction des données pertinentes des logs web ;
3. La conception de modèles permettant de détecter des attaques sur un serveur web ;
4. L'évaluation des performances du modèle en l'utilisant pour détecter des attaques.

1.4 Plan du mémoire

Le mémoire est divisé en cinq parties, la première étant cette introduction. Le chapitre 1 est la présentation des méthodes développées jusqu'à présent pour détecter des anomalies pouvant survenir durant l'exécution d'un programme en utilisant des fichiers de logs. Ce chapitre servira également à introduire les concepts utilisés par ces méthodes. Nous définirons ainsi rapidement ce que sont des automates ou comment fonctionnent les réseaux de neurones. Le but ne sera pas de dresser une liste exhaustive des méthodes traitant le problème que nous cherchons à résoudre mais ce sera de présenter des axes de travail possibles, afin de justifier les choix que nous effectuerons au chapitre suivant.

Le chapitre 2 exposera les modèles d'apprentissage sélectionnés en expliquant les raisons de ces choix. Il servira également à approfondir certains concepts évoqués durant la revue de littérature et à expliquer en détail le fonctionnement de certains algorithmes.

Le chapitre 3, quant à lui, s'intéressera à la réalisation des modèles, à l'ensemble du protocole expérimental et aux résultats obtenus durant leur exécution. Une discussion sur ces derniers s'amorcera dans ce chapitre et elle se prolongera dans la dernière partie qui conclura également

le mémoire. Cette conclusion aura donc pour double fonction de présenter les limites de notre solution ainsi que d'en exposer des axes d'ouverture afin de l'améliorer.

CHAPITRE 2 REVUE DE LITTÉRATURE

Dans ce chapitre, nous allons introduire et définir certains concepts nécessaires à la compréhension du problème et de nos solutions. Nous allons également présenter des travaux existants déjà mis en œuvre pour résoudre des problèmes similaires et qui nous ont inspirés dans nos solutions.

2.1 Les logs

Cette section introduit les logs, donnant un apperçu des informations qu'ils contiennent et de la façon dont ils sont enregistrés.

2.1.1 Les logs du serveur Apache

Les logs du serveur Apache en version 2.4 sont répartis dans différents fichiers en fonction de leur nature. Nous nous intéresserons ici au journal des accès dont les entrées sont regroupées dans un seul fichier. C'est un fichier de texte brut dans lequel chaque entrée est enregistrée sur une et une seule ligne. Le contenu de chaque ligne est cependant organisé selon une nomenclature configurable par l'utilisateur. Étudier cette nomenclature en amont de l'analyse automatique des logs permet de gagner en efficacité (car les informations sont facilement identifiables) et en temps (car l'analyse se fait alors sur des chaînes plus petites). La configuration de base des logs permet d'accéder immédiatement aux informations suivantes :

- L'adresse IP de l'hôte distant. Attention, dans le cas où un serveur mandataire est présent entre le client originel et le serveur, l'adresse affichée sera celle du serveur mandataire.
- L'identité du client. La RFC 1413 ((9)) définit un protocole permettant d'identifier rapidement une machine lors d'une connexion TCP. Ce champ n'est renseigné que si la directive IdentityCheck est positionnée sur On.
- L'identifiant de la personne qui a demandé le document, dans le cas d'un document protégé par un mot de passe.
- L'heure et la date de la réception de la requête.
- La requête du client. C'est une ligne de texte elle aussi paramétrable.
- Le code statut de la réponse retournée au client. La liste des codes statuts et leur signification sont définis au chapitre 10 de la RFC 2616 ((5)).
- La taille de l'objet retourné au client.

Par défaut, la configuration du serveur Apache employé fournit également deux champs supplémentaires :

- Le « Referer », c'est-à-dire le site depuis lequel le client a lancé sa requête. Attention, c'est une information fournie par le client et que celui-ci peut donc modifier à volonté. Il peut également ne pas être renseigné par celui-ci. Par conséquent, il est fortement déconseiller de ce fier à cette information.
- Le « User-Agent », autrement dit le navigateur utilisé par le client pour envoyer sa requête. Encore une fois, c'est une information que le client peut masquer ou altérer.

La version 2.4 du serveur Apache nous permet cependant de modifier les informations enregistrées dans les journaux. Il est possible d'y inclure du texte fixe afin de séparer plus précisément les paramètres dans le but d'aider lors de la lecture ou l'analyse des logs, ainsi que d'autres informations pouvant s'avérer utiles lors de la recherche d'attaques ou d'anomalies :

- Le temps passé par le serveur à effectuer la requête.
- Le nombre de requêtes persistantes (*pipelining*) en cours pour la connexion http actuelle.
- Le port et le nom canoniques du serveur qui a servi la requête.
- Le chemin de la requête.

Il est également possible de ne cibler que certaines opérations en conditionnant l'enregistrement de certains paramètres à la présence d'autres paramètres. On peut par exemple choisir de n'enregistrer que les logs dans le cas où le serveur créé un document (statut HTTP 201), ou au contraire de ne jamais enregistrer les créations de document.

2.1.2 L'écriture des logs

Afin de garantir leur sauvegarde, les logs sont enregistrés dans des fichiers. Cependant, le moment de leur écriture peut varier. Deux scénarios sont possibles : ils peuvent être écrits individuellement dès leur génération, c'est-à-dire pour chaque requête traitée par le serveur, ou bien stockés en mémoire pendant un certain temps avant d'être écrits dans le journal. La seconde méthode peut être avantageuse pour les systèmes limités par des accès disques car ceux-ci seraient alors moins fréquents. Cependant, en cas de dysfonctionnement du serveur, les derniers logs seraient alors définitivement perdus. D'autant plus, l'analyse des logs en temps réel ne serait plus possible. Dans le cas qui nous intéresse, nous écrivons les logs dès leur génération pour éviter le problème précité.

2.2 Les automates

Dans cette section, nous allons définir rapidement ce que sont les automates avant d'expliquer pourquoi leur caractère de machines à états est pertinent dans notre étude. Nous verrons également des techniques permettant de construire un automate à partir de traces d'exécutions.

2.2.1 Quelques définitions

Un *automate fini* (ou plus rigoureusement "automate à nombre fini d'états"), est un modèle mathématique utilisé dans des contextes variés comme la vérification de processus ou la biologie. Cependant, quelle que soit l'utilisation que l'on en fait, un automate rempli toujours la même fonction : il reconnait un *langage*. Les subtilités qui existent entre deux contextes d'utilisation proviennent de ce que l'on considère comme un langage.

Mathématiquement, un automate fini se définit comme étant un quintuplet $\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{F}, \mathcal{I}, \mathcal{T})$ où :

- Σ est un *alphabet* (un ensemble de mot généralement fini et non vide, un mot étant composé de lettres),
- $\mathcal{Q}$ est l'ensemble des *états*,
- $\mathcal{F} : \mathcal{Q} \times \Sigma \times \mathcal{Q}$ est *l'ensemble des transitions*,
- $\mathcal{I} \subseteq \mathcal{Q}$ est l'ensemble des *états initiaux*, généralement réduit à un seul élément,
- $\mathcal{T} \subseteq \mathcal{Q}$ est l'ensemble des *états finaux*.

On représente généralement un automate par un diagramme d'états.

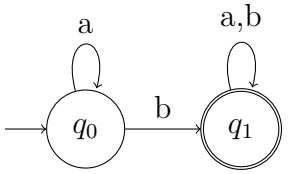

Figure 2.1 Un exemple d'automate

L'automate de la figure 2.1 utilise l'alphabet $\Sigma = \{a, b\}$ possède deux états $\mathcal{Q} = \{q_0, q_1\}$, quatre transitions $\mathcal{F} = \{(q_0, a, q_0), (q_0, b, q_1), (q_1, a, q_1), (q_1, b, q_1)\}$, un état initial $\mathcal{I} = \{q_0\}$ et un état final $\mathcal{F} = \{q_1\}$. Il reconnait le langage des mots constitués des lettres a et b contenant au moins un b.

Comme précisé précédemment, un langage n'est pas nécessairement constitué de mots et de lettres aux sens usuels des termes. Un automate peut par exemple servir à modéliser

le fonctionnement d'un feu tricolore : les couleurs *rouge*, *orange* et *vert* devenant alors les "lettres" sur lesquelles travaille l'automate.

Dans la partie précédent, nous avons introduit les fichiers de logs et les traces d'exécution. Une trace est un ensemble séquentiel de logs, de la même façon qu'un mot est un ensemble séquentiel de lettres. Si l'on considère les traces comme des mots qu'un automate doit reconnaître, il devient alors possible de faire la différence entre des traces correspondant au fonctionnement normal d'un système et celles correspondant à des incidents, à condition d'être capable de générer toutes les traces correspondant à une utilisation normale.

Par ailleurs, en fonctionnant sur les traces d'exécution, il devient possible de créer un algorithme totalement détaché du système d'origine. Cela permet d'ignorer tout de la conception du système ou des détails de son utilisation : c'est l'algorithme qui se chargera de vérifier comment caractériser le fonctionnement normal du système à partir de ses traces d'exécution.

2.2.2 Construire un automate reconnaissant une exécution : kTail

(3) ont proposé une méthode pour construire un automate à partir de ses traces d'exécution. Leur motivation principale était de fournir un modèle formel d'un processus à partir des données qu'il génère. En effet, beaucoup d'outils d'aide au développement et à la maintenance nécessitent l'existence d'un modèle formel du processus à améliorer ((16), (6), (10)) Néanmoins, la génération d'un tel modèle s'avère généralement consommatrice de ressources. Afin d'alléger cette charge et de permettre l'utilisation de nouveaux outils, (3) ont proposé trois méthodes de génération de *machines à états finis*, car elles permettent selon eux d'exprimer efficacement les motifs récurrents au sein d'un processus. La première méthode est un *réseau de neurones* dont la couche cachée est rebouclée sur la couche d'entrée (nous verrons les réseaux de neurones plus en détail au 2.3). L'intérêt de ce bouclage est que la sortie du réseau de neurones ne dépend plus uniquement de l'entrée mais également de l'état actuel (incarné par la couche cachée). Leur seconde méthode consiste à construire un automate à états finis à l'aide d'un algorithme nommé `kTail` dont nous allons développer le fonctionnement par la suite. Leur troisième méthode est d'utiliser des *chaînes de Markov*. L'idée, dans ce troisième cas, est de découvrir quels sont les cheminements résultant de l'exécution normale du processus tout en rajoutant une nuance probabiliste. Une chaîne de Markov est une machine à états, similaire à un automate, mais dont les transitions entre états représentent la probabilité de passer d'un état à un autre. Par ailleurs, la valeur de ces probabilités ne dépend que de l'état sortant, ce qui est une hypothèse raisonnable lorsque l'on analyse des traces d'exécution : l'état futur d'un système ne dépend que de son état actuel.

Revenons désormais à l'algorithme utilisant des automates finis. `kTail` correspond en réalité

à une famille d'algorithmes fonctionnant sur base commune. Chaque implémentation est légèrement différente pour répondre à des spécificités du problème étudié mais le principe général est le même.

L'idée principale derrière tous les algorithmes `kTail` est qu'un état est défini par les comportements futurs qui peuvent survenir à partir de lui. Deux traces peuvent donc partager le même *préfixe* puis diverger, donnant différents comportements futurs à partir une même suite d'évènements initiale. Le *futur*, ou *kFutur*, est défini par la séquence de longueur k, k étant le paramètre de l'algorithme.

`kTail` fonctionne sur des classes d'équivalence : deux préfixes appartiennent à la même classe d'équivalence si les ensembles composés des séquences de longueur k ou moins qui les suivent sont identiques. Un état de l'automate généré correspond à une classe d'équivalence et les transitions correspondent aux classes d'équivalence atteignables depuis cette classe. Notons que l'automate généré n'est pas nécessairement déterministe car un même évènement peut correspondre à deux classes d'équivalence différentes : c'est-à-dire qu'une même lettre (évènement) permettra une transition vers deux états (classes d'équivalences) différents.

La variation proposée par (3) est l'ajout d'une étape de réduction de l'automate, nommée *kInclusion*, en fusionnant deux états qui ont le même *kFutur*. En effet, les automates générés par `kTail` sont des automates dont la complexité (le nombre d'états et de transitions) augmente avec la valeur du paramètre k, rendant leur parcours potentiellement long, particulièrement dans le cas d'un automate fortement non-déterministe. La *réduction* d'un automate consiste à transformer celui-ci en un équivalent moins complexe, retirant des transitions redondantes et fusionnant des états équivalents. La réduction change uniquement la topologie de l'automate et aucunement le langage qu'il reconnait.

2.2.3 Une amélioration de kTail : kBehavior

`kTail` présente un inconvénient majeur : l'algorithme a besoin de traiter toutes les traces d'exécution avant de pouvoir déterminer quels seront les états futurs arrivant après un état donné, car il faut déterminer toutes les classes d'équivalence avant de pouvoir commencer à générer l'automate. Cela empêche une construction incrémentale de l'algorithme. Les utilisations des systèmes pouvant légèrement varier au cours du temps, l'algorithme `kTail` n'est pas capable de réutiliser des connaissances déjà apprises afin de s'adapter aux nouvelles traces générées : il doit tout reconstruire de zéro. Afin de traiter ce problème, (11) ont proposé un nouvel algorithme, `kBehavior`, inspiré de `kTail`.

`kBehavior` n'utilise pas les classes d'équivalence mais directement les évènements. Un état ne

correspond plus à une classe d'équivalence mais à un état dans lequel se trouve le système analysé, ici incarné par une suite d'évènements. Une transition correspond alors à un évènement enregistré dans une trace d'exécution. L'algorithme n'a alors plus besoin de pré-traiter les traces qu'il reçoit. Cependant, il a besoin d'explorer l'automate déjà généré lorsqu'il cherche à apprendre à nouveau : lorsqu'une nouvelle séquence d'évènements doit-être ajoutée à l'automate (de nouveaux états et de nouvelles transitions), il faut s'assurer que cette séquence d'évènements n'est pas déjà présente au sein de l'automate. L'automate généré n'étant généralement que peu non-déterministe, le parcours de l'automate est assez peu coûteux et le surcoût engendré par cette méthode reste raisonnable.

2.3 Les réseaux de neurones

Un des modèles que nous allons proposer étant un réseau de neurones, cette section abordera ce sujet. Nous y donnerons des définitions permettant de manipuler les réseaux de neurones ainsi que des exemples d'études basées sur de tels modèles.

2.3.1 Généralités sur les réseaux de neurones

Le développement des premiers réseaux de neurones remonte aux années 1960. Ce nom "réseaux de neurones" vient de l'organe dont leurs créateurs se sont inspirés : le cerveau. L'idée était qu'afin de rendre les ordinateurs capables d'identifier des motifs aussi rapidement et efficacement que les êtres humains, il fallait s'inspirer de la façon dont ces derniers calculent. Le lien entre un neurone présent dans notre cerveau et celui implémenté dans un ordinateur est finalement très mince mais l'inspiration biologique de cette méthode est néanmoins bien réelle.

Dans la représentation la plus habituelle et répandue, un *réseau de neurones* est un ensemble de *neurones* organisés en *couches* successives. Les neurones de la première couche sont tous reliés à tous les neurones de la seconde couche, eux-même reliés aux neurones de la troisième, etc. Les neurones de la première couche sont surnommés *neurones d'entrée* et les neurones de la dernière couche *neurones de sortie*. Par ailleurs, cette topologie de réseau de neurone s'appelle un *perceptron* (voir figure 2.2). Il existe d'autres formes de réseaux de neurones comme les réseaux de neurones bouclés ((18)), c'est-à-dire dont le graphe de connexions est cyclique, ou une répartition des neurones en grille que nous présenterons plus loin au 2.3.6. Bien que l'objet de cette étude ne soit pas d'établir une telle liste, il faut garder à l'esprit que les topologies de réseaux de neurones sont fondamentalement liées à leurs comportements et sont par conséquent plus ou moins adaptées à certaines utilisations.

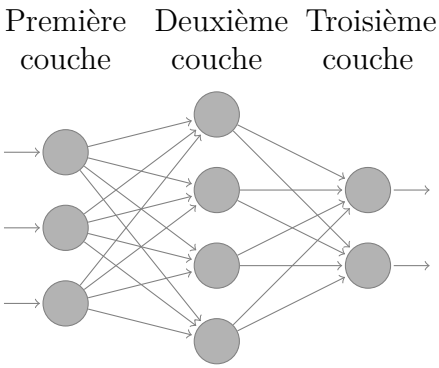

Figure 2.2 Un exemple de perceptron multi-couches

Un neurone, quant à lui, se définit comme une fonction algébrique non linéaire, paramétrée et à valeurs bornées. Les paramètres d'un neurone sont de deux types : le *biais* et les *poids*. Un neurone est donc une *fonction d'activation* à qui l'on passe la combinaison linéaire des paramètres et des variables d'entrée. Les fonctions d'activations les plus répandues sont des fonctions sigmoïdales donc généralement des *arcs tangentes* ou des *tangentes hyperboliques*. Ainsi, un neurone se formalise régulièrement sous la forme $y = th(\omega_0 + \sum_{i=1}^{n} \omega_i x_i)$, y étant la sortie du neurone, $\{\omega_i\}_{0 \leq i \leq n}$ les paramètres et $\{x_i\}_{0 \leq i \leq n}$ les variables.

Si on extrapole cette définition, on peut se représenter une couche de neurones comme une fonction prenant en variable d'entrée un vecteur et retournant un vecteur en sortie, de dimension potentiellement différente. La succession de ces couches n'est alors plus qu'une composition de fonctions. Un réseau de neurones n'est finalement qu'une grosse fonction mathématique possédant énormément de paramètres à estimer.

Par ailleurs, notons que (7) a montré que les perceptions étaient un approximateur universel. Autrement-dit, en fournissant suffisamment de couches cachées, et suffisamment de données pour effectuer l'apprentissage, un réseau de neurones multi-couches est capable d'approcher n'importe quelle fonction avec la précision désirée. Cependant, rajouter des couches cachées présente plusieurs inconvénients comme un apprentissage plus lent ou un modèle qui perd en généralisation. Le plus grand danger en apprentissage machine est de réaliser un modèle trop complexe, qui colle très bien aux données d'apprentissage mais qui effectue beaucoup d'erreurs dans le cas général. Ce danger se nomme le *surapprentissage*.

2.3.2 Apprentissage

Comme cela vient d'être évoqué, un réseau de neurones, comme tout modèle d'apprentissage machine, a besoin d'apprendre, et a besoin de données pour ce faire. On sépare généralement

les données dont on dispose en deux groupes : les *données d'apprentissage* et les *données de validation*. Les données d'apprentissage servent à déterminer les paramètres des neurones et les données de validation servent à vérifier que le modèle reste performant dans le cas général et ne colle pas trop aux données d'apprentissage afin d'éviter le surapprentissage.

L'apprentissage d'un réseau de neurones dépend de sa topologie. La méthode que nous allons présenter est celle associée aux perceptrons multicouches : la *rétro-propagation*. Une donnée d'apprentissage (et donc de validation également) contient les valeurs présentes dans chaque donnée que l'on fournira à notre perceptron mais contient également les valeurs qu'est censé rendre le réseau après calcul. Ainsi, on est capable de mesurer la différence entre la valeur attendue et celle obtenue. On détermine ensuite quelle est la contribution de chaque paramètre à l'*erreur* pour tous les paramètres de tous les neurones du réseau. Cette opération ne peut commencer que par la fin du réseau étant donné que c'est elle qui contient initialement l'erreur. On remonte ensuite le long des connexions du réseau pour répercuter cette erreur et l'ajustement des paramètres, d'où le terme de rétro-propagation.

2.3.3 Apprentissage supervisé ou non ?

En machine-learning, il existe deux types d'apprentissage : l'apprentissage supervisé et l'apprentissage non-supervisé. Selon que les données que l'on fournit sont étiquetées ou non, on se trouve dans l'une ou l'autre des deux formes. Lorsque l'on fait du *clusturing* par exemple, ou *aggrégation* en français, on ne connait pas à l'avance quelles sont les classes des données que l'on manipule, c'est d'ailleurs souvent ce que l'on cherche à déterminer : c'est de l'apprentissage non-supervisé. Dans le cas de l'apprentissage supervisé, le but est souvent de se servir de données connues pour prédire le comportement ou les valeurs de nouvelles données. Certaines méthodes sont dédiées à un certain type d'apprentissage. Les réseaux de neurones, selon leur topologie, les données qu'on leur fourni, etc. permettent de couvrir l'une ou l'autre des deux formes d'apprentissage.

2.3.4 Competitive Learning

L'apprentissage compétitif est une forme d'apprentissage non supervisé. Le principe de cette forme d'apprentissage est que les neurones sont tous en compétition pour répondre à un stimulus donné. Le neurone qui répond le mieux à ce stimulus (qui a la réponse la plus proche de celle attendue, au sens de la distance euclidienne dans la majorité des cas) est déclaré *vainqueur*. Le neurone vainqueur voit alors ses paramètres ajustés afin qu'il réponde encore mieux à un stimulus similaire. Les ajustements apportés à ses paramètres se répercutent ensuite sur ses voisins, au sens topologique, mais avec une intensité moindre, puis à leurs

voisins, etc., jusqu'à recouvrir la totalité du réseau.

2.3.5 Des machines à vecteur de support

(15) se sont concentrés sur de la prédiction d'erreurs en analysant des logs système. Afin de réaliser leur objectif, ils ont retenu trois implémentations différentes des machines à vecteur de support : un perceptron multi-couches, une fonction de base radiale et des noyaux linéaires.

Une machine à vecteur de support (Support Vector Marchine – SVM) est un algorithme d'apprentissage supervisé travaillant sur des vecteurs de dimension potentiellement élevée, répartis en deux classes. Le but d'un SVM est de séparer les deux classes de la façon la plus optimale possible. Pour ce faire, le SVM cherche à diviser l'espace en deux, un hyperplan étant la frontière : tout ce qui est d'un côté de l'hyperplan appartient à une classe et tout ce qui est de l'autre côté appartient à une autre classe comme illustré sur la figure 2.3.

Une fois l'hyperplan appris, les nouvelles données dont la classe n'est pas connue peuvent facilement être répertoriées en fonction de leur position par rapport à l'hyperplan.

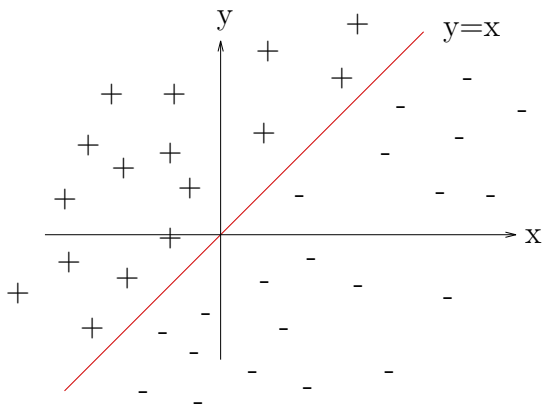

Figure 2.3 L'hyperplan d'équation $y = x$ sépare les deux classes "plus" et moins"

2.3.6 Des réseaux de neurones pour faire de l'agrégation

Les réseaux de neurones ont notamment fait l'objet d'une étude récente de (18) cherchant à différencier les robots d'indexations légitimes (tels que MSNBot, GoogleBot ou YahooBot) de robots malicieux, dont le but est d'effectuer une attaque de type déni de service (Denial of Service – DoS). Lors d'une attaque DoS, l'attaquant va envoyer beaucoup de requêtes au serveur afin de le surcharger, le ralentir ou même le rendre inopérant. Un robot d'indexation va, quant à lui, parcourir une page web, récupérer les liens hypertexte présents sur cette page et suivre ces liens. De cette façon, il va parcourir le site web, et consommer des ressources

sur le serveur mais à des fins d'indexation. Il est possible pour une attaque d'imiter ce comportement et donc de passer outre des contre-mesures destinées à se prémunir contre une attaque DoS moins sophistiquée.

(18) propose l'utilisation de deux algorithmes d'apprentissage non supervisé fondés sur des réseaux de neurones afin d'identifier les différents clients se connectant sur un serveur web dans le but de différencier les attaques d'utilisations normales : *Self-Organizing Map (SOM)* et *Modified Adaptive Resonance Theory 2 (Modified ART2)*.

L'algorithme `SOM` (Carte auto adaptative en français) fait en sorte que des neurones réagissent de façon similaire à des motifs proches. Encore une fois, on retrouve un lien assez fort avec le fonctionnement du cerveau dans lequel deux stimuli de même nature excitent les mêmes régions du cerveau. Dans la pratique, c'est un algorithme d'apprentissage compétitif. `SOM` est particulièrement renommé pour être robuste aux anomalies statistiques et pour agréger des espaces d'entrées à dimension élevée dans des résultats représentables dans un espace à deux dimensions.

Par ailleurs, la topologie du réseau est différente de la topologie habituelle : les neurones sont organisés en grilles où chacun est connecté à ses plus proches voisins (figure 2.4) ;

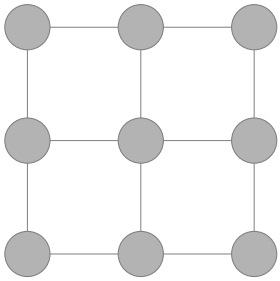

Figure 2.4 Topologie d'un réseau de neurones en grille

L'algorithme `ART2` est également une forme d'apprentissage compétitif mais qui rajoute la règle du *Winner-take-all*. Dans cette situation, suivant un stimulus d'entrée, le neurone vainqueur est celui dont la réponse minimise la distance euclidienne à la réponse attendue. `ART2` est particulièrement performant pour son équilibre entre l'apprentissage de nouveaux motifs et la conservation d'anciens et pour sa capacité à identifier des agrégats sous-représentés.

`ART2` est composé de deux couches de neurones. La première couche correspond au champ de comparaison et la seconde au champ d'identification. A cela s'ajoute un *paramètre de vigilance* et un module de remise à zéro. Les neurones de la couche de comparaison reçoivent les valeurs d'entrées et les transmettent aux neurones de la couche d'identification répondant le mieux à ce stimulus (le *Winner-take-all*). Cependant, avant de réaliser l'apprentissage, le

module de remise-à-zéro compare la valeur retournée par ce neurone au seuil de vigilance. Si le seuil est dépassé, d'autres neurones sont considérés. Si aucun neurone n'est trouvé, un nouveau neurone est ajouté au champ d'identification et ses paramètres sont ajustés.

2.4 D'autres techniques d'analyse de log

La littérature regorge de modèles différents pour détecter des anomalies. Nous allons en présenter quelques autres néanmoins non retenus dans nos solutions mais qui illustrent la grande variété d'outils à notre disposition et dont les spécificités s'accordent à celles des problèmes qu'ils traitent.

2.4.1 Utilisation d'arbres de décision

(13) ont cherché à classer les erreurs survenant lors de l'exécution d'un système à partir des symptômes qu'elles laissent dans les logs. La majorité de leur travail consiste en l'extraction d'informations à partir des logs, supposés être de simples lignes de texte. Cependant, après avoir extrait les informations, ils décident de construire un arbre de décision afin de traiter les données ultérieures.

Un *arbre de décision*, illustré à la figure 2.5, est une forme d'apprentissage supervisé dans laquelle on cherche à classer des données en fonction d'observations. Une donnée est découpée en attributs possédant un nombre fini de valeurs. Ces attributs se retrouvent sur les embranchements de l'arbre de décision, et chaque branche correspond à une valeur différente de l'attribut. Une fois que l'arbre est construit, il suffit de regarder l'attribut correspondant à l'embranchement auquel on est rendu, de suivre la valeur correspondant à cet attribut puis recommencer à chaque embranchement jusqu'à arriver à une feuille. Une feuille correspond à la classe de la donnée.

Un arbre de décision pourrait se construire en choisissant les attributs au hasard et serait parfaitement valide. Il ne serait néanmoins que peu performant car il pourrait privilégier des attributs qui ne différencient que peu les données. Afin de construire l'arbre de décision efficacement, il existe plusieurs approches. Celle retenue par (13) est l'algorithme `C4.5` de (17).

Pour déterminer quel est l'attribut le plus pertinent, (17) fait appel à un mécanisme issu de la théorie de l'information : *l'entropie de Shannon*.

Considérons un jeu de données $S = \{s_1, ..., s_N\}$ dans lequel chaque donnée possède P attributs $\{A_j\}_{1 \leq j \leq P}$. Chaque donnée s_i appartient à une classe d_k à valeur dans un ensemble

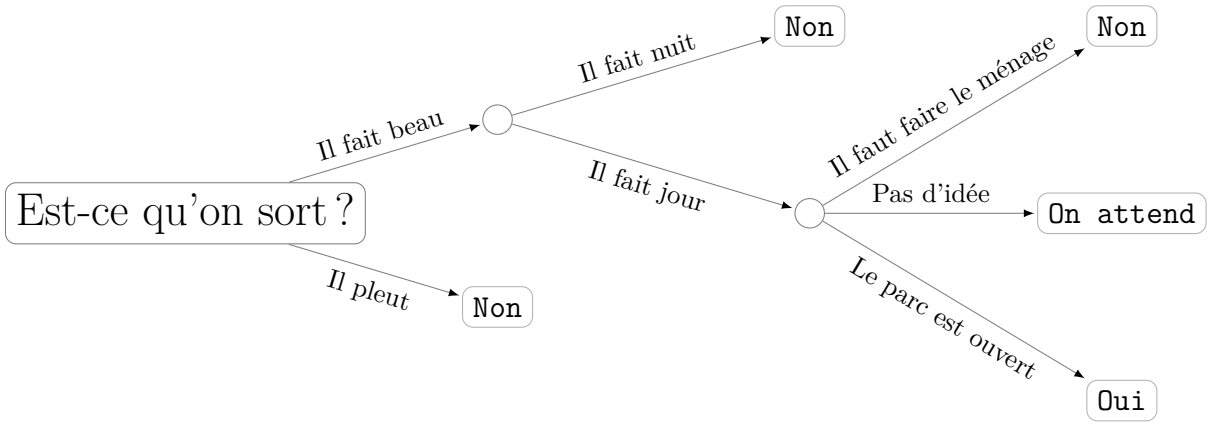

Figure 2.5 Un arbre de décision

$\{d_k\}_{1 \leq k \leq M}$. Chaque classe d_k contient m_k membres.

Commençons par calculer l'entropie initiale. Elle est liée à la répartition des données dans les différentes classes et s'exprime ainsi :

$$E_I = \sum_{k=1}^{M} -\frac{m_k}{N} log(\frac{m_k}{N})$$

Le but est ensuite de choisir l'attribut qui différencie le mieux les données, c'est-à-dire diminuant le plus l'entropie résultant du choix de cet attribut. Pour cela, il faut calculer l'entropie liée à chaque attribut. L'attribut A_j peut prendre N_j valeurs différentes possibles dans l'ensemble $\{a_{j,l}\}_{1 \leq l \leq N_j}$. Chaque valeur $a_{j,l}$ correspond à un sous-ensemble de l'espace de données initial de taille $M_{j,l}$. Dans cet ensemble, chaque classe s_i est représentée m_i fois. L'entropie correspondant au choix de cet attribut est alors :

$$E(a_{j,l}) = \sum_{i=1}^{M} -\frac{m_i}{M_{j,l}} log(\frac{m_i}{M_{j,l}})$$

On fait ensuite la moyenne sur toutes les valeurs de l'attribut :

$$Reste(A_j) = \sum_{l=1}^{N_j} \frac{M_{j,l}}{N} E(a_{j,l})$$

Et on calcule enfin le gain :

$$Gain(\{a_{j,i}\}_{1 \leq i \leq p}) = E_I - Reste(\{a_{j,i}\}_{1 \leq i \leq p})$$

On répète l'opération pour tous les attributs et on choisit alors l'attribut maximisant ce gain.

2.4.2 Un classifieur bayésien naïf

(14) se sont fondés sur l'étude de (2) concluant que la plupart des erreurs rencontrés par les systèmes sont causées par des fautes récurrentes. Leur hypothèse est donc qu'il est intéressant d'apprendre d'erreurs déjà survenues dans le passé afin de les diagnostiquer le plus rapidement possible lorsqu'elles réapparaissent. Pour ce faire, (14) combinent deux modèles : un arbre de décision basé sur l'algorithme `C4.5` et un classifieur bayésien naïf.

Un classifieur bayésien naïf est un modèle probabiliste fondé sur une hypothèse forte : *les attributs d'une donnée sont tous deux à deux indépendants* (d'où son caractère naïf).

Considérons une donnée possédant n attributs de valeurs $F_1, ..., F_n$. En appliquant la formule de Bayes à la probabilité conditionnelle d'appartenir à une classe C sachant les valeurs des attributs, on obtient :

$$p(C|F_1, ..., F_n) = \frac{p(C)p(F_1, ..., F_n|C)}{p(F_1, ..., F_n)}$$

Les valeurs des $F_1, ..., F_n$ sont données et la valeur du dénominateur ne nous intéresse donc pas puisqu'il est constant. Le numérateur se développe ainsi :

$$p(C)p(F_1, ..., F_n|C) = p(C)p(F_1|C)p(F_2|C, F_1)...p(F_n|C, F_1, ..., F_{n-1})$$

Or, l'hypothèse du réseau bayésien naïf est que les $F_1, ...F_n$ sont tous deux à deux indépendants. Ainsi :

$$p(F_1, ..., F_n|C) = \prod_{i=1}^{n} p(F_i|C)$$

Donc, Z représentant les probabilités des $F_1, ..., F_n$ qui sont, rappelons le, constantes :

$$p(C|F_1, ..., F_n) = \frac{1}{Z}p(C)\prod_{i=1}^{n} p(F_i|C)$$

L'inconvénient du classifieur bayésien naïf, dans ce contexte de diagnostique de fautes, est qu'il permet d'identifier les fautes potentielles mais pas de les relier à des traces de logs en particulier. Afin de remédier à ce problème, (14) doublent le classifieur bayésien naïf d'un arbre de décision dont le fonctionnement a été évoqué au 2.4.1

CHAPITRE 3 LE MODÈLE

3.1 Deux modèles très différents mais pas incompatibles

L'objectif de ce mémoire est, rappelons-le, de transposer des techniques de détection d'anomalies afin d'identifier des attaques. Pour ce faire, nous allons considérer une méthode déterministe et une méthode probabiliste.

Du côté déterministe, nous allons choisir un automate fini. Il fonctionnera sur des traces d'exécutions et servira à déterminer si une trace correspond à une attaque ou non. L'hypothèse est la suivante : si une trace n'est pas reconnue par l'automate, c'est qu'elle n'a jamais été rencontrée lorsque l'on a entrainé notre modèle. Si on entraîne notre modèle sur l'ensemble des utilisations possibles du système, toute trace non reconnue correspond à une anomalie, et donc potentiellement à une attaque.

Contrairement au cas déterministe où le nombre de modèles pertinents semble assez réduit, selon le contenu de la littérature, une approche probabiliste offre beaucoup de perspectives avec une grande variété de modèles. Le but de notre étude n'est pas de déterminer lequel de ces modèles est le plus performant. Il ne s'agit pas de les comparer, mais d'en sélectionner un. Notre choix s'est donc porté sur le *perceptron multicouches*. D'après (7) un perceptron multicouches est un approximateur universel. Il semble donc être un bon point de départ pour notre étude, une sorte de médiane des modèles probabilistes. La fonction que l'on va chercher à réaliser ici est : étant donnée un certain nombre de paramètres, suis-je la cible d'une attaque ?

Un perceptron multicouches ne peut pas prendre des traces d'exécution en entrée, il faut lui fournir des nombres. La première étape est donc d'extraire des caractéristiques de ces traces, de les quatifier et enfin de les donner au perceptron. Celui-ci calculera alors la probabilité qu'une attaque soit en court. Contrairement à l'automate qui donne une réponse binaire, le perceptron rend une probabilité, ce qui apporte une nuance supplémentaire. Cependant, il n'est pas possible d'effectuer d'hypothèses sur son fonctionnement car la détermination de ses paramètres dépend des données sur lesquelles il apprend.

Notons une autre différence entre nos deux modèles qui concerne leur utilisation, dans l'usage cette fois. L'automate dit clairement ce qu'il reconnait et ne reconnait pas, et ce qu'il ne reconnait pas est une attaque. Cependant, de part sa nature binaire, comme nous l'avons évoqué, il n'est pas capable de diagnostiquer une attaque, dans le cas où il en identifie une. Le perceptron, quant à lui, peut être utilisé pour apporter plus de finesse dans l'identification

de l'anomalie. Il n'est pas simplement capable de détecter une attaque mais il peut également la qualifier, en déterminer la nature. Nous reviendrons sur ces points plus en détail par la suite mais il est important de noter, dès à présent, ce contraste qui a été un critère de choix lors de l'adoption de ces deux modèles.

Faisons un petit aparté qui dépasse le cadre de notre étude mais qui peut aisément en être un prolongement. Les deux modèles évoqués, même s'ils sont utilisés dans le même but - détecter des attaques - peuvent évoluer en collaboration. On peut penser intuitivement que deux modèles valent mieux qu'un mais ce n'est pas nécessairement vrai. Un perceptron et un SVM par exemple vont globalement utiliser les mêmes paramètres pour retourner le même résultat. Ils sont redondants et les ressources dépensées à les apprendre et à les utiliser sont alors gâchées. Dans notre cas, le perceptron peut aisément incorporer les résultats de l'automate dans ses paramètres d'entrée ou bien un administrateur peut décider de surveiller différentes attaques ou différents points du systèmes avec ces deux modèles. Encore une fois, il ne s'agira pas ici de combiner ces deux modèles pour en extraire de l'information supplémentaire mais cette compatibilité s'est avérée séduisante et a présenté un argument supplémentaire dans notre sélection.

Une fois le choix des modèles effectué, il faut commencer par récupérer et formater les données de façon à ce que les modèles puissent les utiliser.

3.2 Identifier les traces et extraire des données

Dans les expérimentations que nous avons effectuées, nous avons fait le choix d'utiliser un serveur web Apache en version 2.4.20. Nous reviendrons plus tard sur ce choix mais il est nécessaire de le mentionner afin d'expliquer quelles caractéristiques il est possible d'extraire de ce serveur.

Comme nous l'avons vu au chapitre précédent, les logs d'Apache sont stockés sous forme de lignes dans un fichier et selon un format particulier. Afin d'en extraire des informations, il n'est donc pas nécessaire de faire appel à des techniques d'extractions de données comme dans les études réalisées par (19), (20), (14) ou d'autres. Cependant, ces études sont fondées sur l'hypothèse que ces logs sont des chaînes de caractères sans structure particulière. Dans le cas d'un serveur Apache, ce n'est pas le cas. Il suffit de suivre l'organisation des données fournie par la documentation d'Apache et de découper les chaînes de caractères (dans notre cas ce fût avec des expressions régulières) afin d'accéder aux informations désirées.

Cependant, ces données ainsi extraites sont encore brutes et ont besoin d'être affinées pour être utilisées par les deux modèles. Chacun des deux modèles nécessite des opérations parti-

culières car ses paramètres d'entrée sont très différents.

3.2.1 Le cas de l'automate

Rappelons-le : un automate fonctionne sur des lettres. Dans l'utilisation que l'on souhaite en faire, l'automate doit identifier des séquences d'exécution. Autrement dit : quelles sont les successions de pages web que les clients consultent. Cette information se trouve dans la *requête* enregistrée dans le log. Attention, certaines requêtes peuvent utiliser des paramètres qui apparaissent dans le corps de la requête (avec la méthode GET par exemple). Dans ce cas, il est nécessaire de les éliminer afin d'éviter d'avoir un automate trop spécifique.

Une fois que l'on a extrait les requêtes de chaque log, il faut les agréger en *mots*, c'est à dire en ensembles séquentiels de lettres, afin que ces mots correspondent à des ensembles d'états et de transitions que le modèle devra apprendre.

Tableau 3.1 Des traces d'exécution

192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /index.php HTTP/1.1" 200 870 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /notes.php HTTP/1.1" 200 656 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /notes.php ?id=1 HTTP/1.1" 200 610 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /index.php HTTP/1.1" 200 814 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /addnote.php HTTP/1.1" 302 793 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /login.php HTTP/1.1" 200 791 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /index.php HTTP/1.1" 200 814 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /login.php HTTP/1.1" 200 791 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /index.php HTTP/1.1" 200 814 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /notes.php HTTP/1.1" 200 656 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /notes.php ?id=2 HTTP/1.1" 200 619 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /index.php HTTP/1.1" 200 814 "-" "-"
192.168.100.1 - - [05/Jun/2016 :20 :15 :23 -0700] "GET /index.php HTTP/1.1" 200 814 "-" "-"

Prenons le cas du tableau 3.1. La première opération à effectuer est de découper le fichier de log pour isoler les différentes traces. Pour ce faire, il faut isoler les logs générés par différents clients (ici identifiés de façon unique par leur adresse IP mais dans un cadre plus général, cette hypothèse peut devenir discutable, comme dans une connexion derrière un NAT par exemple) puis isoler les différentes traces. Ce découpage est, à vrai dire, optionnel dans le cas de l'algorithme d'apprentissage que nous avons sélectionné mais il permet de gagner en efficacité et en granularité lors de la phase de détection d'attaque et du diagnostique ultérieur. Dans notre cas particulier, une trace est identifiée comme étant le cheminement effectué par un client sur le site entre deux consultations de la page d'index. Partant de cette hypothèse, on peut découper notre exemple précédent comme illustré sur le tableau 3.2.

Tableau 3.2 Cinq traces identifiées

index.php → notes.php → notes.php
index.php → addnote.php → notes.php
index.php → login.php
index.php → notes.php → notes.php
index.php → index.php

On obtient alors deux traces d'exécutions que notre automate va apprendre avec l'algorithme que nous présenterons par la suite.

3.2.2 Le cas du perceptron

Mais restons pour l'instant sur le format des données reçues par les deux modèles. Le perceptron ne peut pas, nous allons le voir, utiliser les mêmes données que l'automate pour réaliser son apprentissage ou pour son fonctionnement. Les données brutes sont les mêmes - les logs - mais ces derniers ne contiennent pas que des requêtes. La complexité supérieure du perceptron va nous permettre de tirer profit d'autres données contenues dans les fichiers de log afin d'extraire des informations différentes.

Un perceptron prend un vecteur en entrée, c'est-à-dire un ensemble de nombres. Contrairement à l'automate, il n'est pas possible de lui transmettre une chaîne de caractères brutes. Il est cependant possible de multiplier les paramètres d'entrée afin de fournir plus d'informations au modèle et d'obtenir des données plus précises. Cette multiplication présente néanmoins le risque d'augmenter la complexité du perceptron, et donc d'augmenter les temps de calcul, sans pour autant apporter des informations utiles. Les paramètres d'entrée du perceptron qui ont été sélectionnés sont donnés et justifiés à la section 3.4.1.

3.3 Un automate éduqué pour reconnaître ce qu'il a déjà vu

Formater les données sert simplement à les préparer pour les deux étapes suivantes : l'apprentissage et l'exécution. Nous allons traiter l'apprentissage de l'automate dans les prochaines parties.

3.3.1 L'apprentissage

Le principe de base de la construction de l'automate semble simple et intuitif. Initialement, l'automate ne reconnaît aucun langage ; il est vide. Il est construit de façon incrémentielle.

S'il reconnaît une trace, il ne se passe rien. Dans le cas contraire, il devra modifier sa structure pour reconnaître cette trace.

Néanmoins, une implémentation directe d'une telle procédure va générer un automate très lourd, possédant beaucoup d'états et de transitions dont la plupart seront assurément redondants. L'enjeu est ici de construire un automate capable non seulement de reconnaître des traces d'exécution mais également d'identifier des boucles pouvant être effectuées un nombre de fois arbitraire par un client, tout en garantissant une complexité minimale. Ainsi, il devra être capable d'identifier des portions de trace déjà reconnues afin de ne pas générer de portion redondante.

Par ailleurs, afin de garantir une évolutivité future et une certaine robustesse face à des changements mineurs dans l'utilisation du système, celui-ci devrait se construire au fur et à mesure de la réception des traces : un algorithme qui nécessite de lire l'ensemble des traces avant de générer l'automate présente l'inconvénient majeur de ne pas s'adapter à l'apparition de nouvelles traces.

Comme évoqué dans la revue de littérature, un algorithme répondant à ces attentes a été développé par (11). Il se nomme `kBehavior` et son fonctionnement fait l'objet de la section suivante.

3.3.2 kBehavior

(11) ont développé, dans un contexte de *composants pris sur étagère* (Commercial of the shelf - COTS), des techniques pour inférer le fonctionnement et l'interopérabilité de ces composants. En effet, certaines erreurs peuvent survenir et s'avérer handicapantes pour les développeurs et les testeurs n'ayant pas toujours accès au code source de ces outils. Il devient alors difficile de diagnostiquer des problèmes dont ils seraient la source. Les interactions entre ces composants sont apprises par un automate fini et permettent aux développeurs de trouver la source de ces problèmes. La nature même du problème empêche l'automate d'avoir la moindre connaissance, à priori, sur le système qu'il modélise, ce qui est particulièrement intéressant dans notre cas.

L'algorithme qui réalise la construction de cet automate, à partir de traces d'exécution, a été nommé `kBehavior`. Il est une déclinaison de l'algorithme `kTail`, ainsi que nous l'avions présenté au chapitre précédent. Étudions son fonctionnement.

`kBehavior` construit, de façon incrémentielle, un automate à partir de traces d'exécution. A la première itération, l'automate n'existe pas. `kBehavior` prend donc la première trace S et la découpe de la façon suivante : $S = S_{préfixe}S_{suffixe}$ avec $|S_{préfixe}| = k$, k étant un paramètre à

définir par l'utilisateur. `kBehavior` génère alors un automate linéaire correspondant à la trace $S_{préfixe}$ et s'applique récursivement avec l'automate qu'il vient de créer et la trace restante $S_{suffixe}$.

Dans le cas où l'automate n'est pas vide (c'est-à-dire après la première itération de l'algorithme) et que l'on veut y ajouter la trace S, on découpe S sous la forme $S = S_{préfixe}S_{suffixe}$, $S_{préfixe}$ étant le préfixe le plus long partant de l'état initial de l'automate. $S_{préfixe}$ est donc une sous-trace reconnue par l'automate. Notons q le dernier état de $S_{préfixe}$. On va alors insérer $S_{suffixe}$ dans l'automate. Pour cela, `kBehavior` va chercher une sous-séquence de $S_{suffixe}$ générée par un sous-automate A' de A. Seules les sous-séquences de longueur supérieure à k sont considérées. Un autre paramètre k_p est employé ici. Il définit la taille minimale satisfaisante des sous-séquences recherchées : l'algorithme s'arrête lorsqu'il a trouvé une sous-séquence de longueur supérieure à k_p.

Ces opérations méritent de plus amples explications. `kBehavior` fait particulièrement attention à une chose : reconnaître une trace déjà apprise. Il n'est pas rare que la même trace apparaisse plusieurs fois dans l'espace d'apprentissage, ou que deux traces différentes commencent de la même façon. Il est alors inutile de travailler sur ces traces : c'est pour cela qu'on identifie les sous-traces $S_{préfixe}$ qui correspondent à des cas déjà traités. Cependant, la trace $S_{suffixe}$ inconnue ne commence pas à un état initial de l'automate. Si on ne conservait pas l'endroit à partir duquel l'automate avait cessé de reconnaître la trace, on génèrerait des séquences d'exécution ne correspondant à aucun cas. L'état q est gardé en mémoire pour cette raison.

Une fois que l'on a obtenu notre sous-trace $S_{suffixe}$, on va chercher à la découper à nouveau en trois sous-traces : $S_{suffixe} = S_{pre}S'S_{suite}$. La sous-séquence S' est la séquence générée par l'automate A' de A indiqué aux paragraphes ci-dessus. La raison est la suivante. Notre sous-trace contient une partie inconnue puisqu'elle n'a pas été totalement générée par l'automate (sinon, $S_{préfixe}$ correspondrait à l'intégralité de la trace). Cependant il y a peut-être une partie reconnue S' plus loin dans $S_{suffixe}$, et si c'est le cas, cela veut dire que la partie S_{pre} comprise avant S' correspond à une boucle, ou à un ensemble d'états et de transitions connectant des parties de l'automate auparavant indépendantes. Il faut que cette séquence S' ait une certaine longueur pour qu'elle soit assez significative dans l'automate (cette longueur est k) mais on ne cherche pas nécessairement une longueur maximale (k_p est cette longueur satisfaisante). Enfin, S_{suite} est la trace restant non encore étudiée et sur laquelle on appliquera à nouveau l'algorithme.

Reprenons désormais le fonctionnement de `kBehavior`. Dans le cas où aucune séquence S' n'est trouvée, un nouvel automate générant $S_{suffixe}$ est généré. Si $|S_{suffixe}| < 2k$, l'automate

généré correspond à la trace $S_{suffixe}$. Si $|S_{suffixe}| \geq 2k$, l'automate généré est obtenu en exécutant récursivement `kBehavior` avec une trace $S_{suffixe}$ et un automate vide. Le nouvel automate est alors ajouté en le fusionnant avec son automate père à la position q.

Au contraire, dans le cas où une sous-séquence S' de $S_{suffixe}$ est trouvée, on note q_i l'état initial de l'automate A' et S_{pre} le préfixe de $S_{suffixe}$ (on peut donc écrire $S_{suffixe}$ sous la forme $S_{suffixe} = S_{préfixe}S'S_{suite}$). On distingue alors deux cas :

Si $q_i = q$, on ajoute un nouvel état q', on bascule toutes les arrêtes incidentes à q vers q', on relie q' à q par le mot vide et q' prend le rôle de q (q' devient par exemple le nouvel état initial si c'était le cas de q. Dans le cas où q_i est différent de q, on passe directement à l'étape suivante.

Il faut ensuite produire un automate A_{pre} qui génère S_{pre} de la même façon que lorsqu'une sous-séquence n'est pas trouvée. Il faut ensuite insérer A_{pre} dans A en connectant q à l'état initial de A_pre et les états finaux de A_{pre} à q_i. q devient alors l'état atteint par la sous-séquence obtenue précédemment et S devient S_{suite}. On exécute alors récursivement `kBehavior` sur S_{suite}.

Afin d'améliorer la compréhension de cet algorithme, le voici transcrit sous forme de pseudo-code. Dans un soucis de concision, la partie concernant le k_p a été omise, complexifiant inutilement la compréhension de l'algorithme. Certaines procédures sont utilisées dans l'algorithme et leurs fonctions sont précisées après celui-ci.

Algorithm 1 `kBehavior`

fonction kBehavior(fsa, S, k) ▷ Apprentissage de la trace S à l'automate fsa
  si fsa est vide **alors**
    $S_{pre} \leftarrow$ préfixe de longueur k de S
    $S \leftarrow$ reste de S
    $fsa \leftarrow$ automate linéaire générant S_{pre}
    $q \leftarrow$ état final de fsa
  sinon
    $q \leftarrow$ état initial de fsa
  fin si
  répéter
    $S_{pre} \leftarrow$ plus long préfixe de S généré par fsa à partir de q
    $S \leftarrow$ reste de S
    $q \leftarrow$ état atteint par S_{pre}
    Identifier une sous-séquence S' de S générée par fsa de longueur minimale k générée par un sous-automate $subfsa$.
    si Une sous-séquence $S' = S(j, l)$ est trouvée **alors**
      $q_i \leftarrow$ état initial de $subfsa$
      $S_{pre} \leftarrow$ préfixe de S de taille $j - 1$
      si $q = q_i$ **alors** ▷ Cela évite des boucles n'ayant aucune valeur sémantique
        Ajouter un nouvel état q'
        Changer toutes les transitions atteignant q vers q'
        Ajouter une transition de q' vers q avec le mot vide
      fin si
      si $|S| < 2k$ **alors**
        $fsa' \leftarrow$ automate linéaire généré par S_{pre}
      sinon
        $fsa' \leftarrow kBehavior(vide, S_{pre}, k)$
      fin si
      $fsa \leftarrow fusion(fsa, q, q_i, fsa')$
      $q \leftarrow$ état atteint par la trace $S_{pre} + S$
      $S \leftarrow S(l + 1, |S|)$
    sinon
      si $|S| < 2k$ **alors**
        $fsa' \leftarrow$ automate linéaire généré par S
      sinon
        $fsa' \leftarrow kBehavior(vide, S, k)$
      fin si
      $fsa \leftarrow fusion(fsa, q, fsa')$
      $S \leftarrow vide$
    fin si
  jusqu'à S est vide
fin fonction

La fonction $fusion(fsa, S, fsa')$ fusionne les automates fsa et fsa' en fusionnant l'état q de fsa avec l'état initial de fsa'. La fonction $fusion(fsa, q_1, q_2, fsa')$ fusionne les automates fsa et fsa' en fusionnant l'état q_1 de fsa avec l'état initial de fsa' et l'état q_2 de fsa avec les états finaux de fsa'. Si la trace S se développe de la façon suivante $S = s_0 s_1 ... s_j ... s_l ... s_{n-1} s_n$, $S(j, l)$ correspond à la sous-séquence $s_j ... s_l$.

Maintenant que le principe algorithmique a été exposé, illustrons le par un exemple. Pour cela, reprenons les traces extraites au tableau 3.2, ajoutons-en quelques unes et construisons l'automate reconnaissant ces traces. Prenons le cas $k = 2$ et également $k_p = 2$ pour simplifier les choses. Par souci d'économie de place, nous ne dessinerons pas l'extension *.php* des fichiers sur l'automate.

Initialement, l'automate est vide. La première trace que l'on considère est la trace $S = index.php \rightarrow billets.php \rightarrow billets.php$. L'automate étant vide, on définit $S_{pre} = index.php \rightarrow billets.php$" et $S = billets.php$. On construit l'automate linéaire générant $Spre$ et on se place à la fin de celui-ci, c'est à dire $q = 2$;

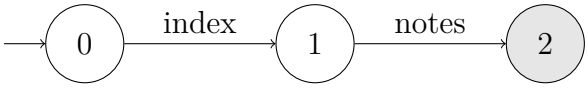

Figure 3.1 La génération du premier automate

A l'issue de cette étape, on a donc $S = billets.php$ et $q = 2$. On est donc dans le cas où aucune séquence n'est trouvée. Etant donné que $|S| < 4$, on fusionne notre automate avec l'automate linéaire générant S. A l'issue de cette opération, S est vide : $S = \emptyset$. `kBehavior` se termine donc après cette étape.

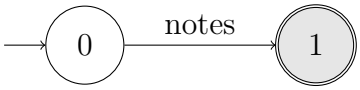

Figure 3.2 L'automate à fusionner

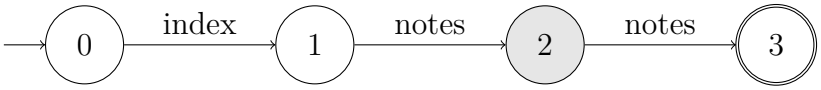

Figure 3.3 Résultat de `kBehavior` sur la première trace

Considérons désormais la seconde trace que l'on va rajouter, si besoin, à l'automate. On initialise donc les paramètres $S = index.php \rightarrow ajouter\_billet.php \rightarrow login.php$ et $q = 0$.

On identifie le plus long préfixe de la trace et on met à jour q : $S_{pre} = index.php$, $S = ajouter\_billet.php \rightarrow login.php$ et $q = 1$. On cherche alors à identifier une sous-séquence de longueur 2 déjà présente dans l'automate, sans succès. On crée alors l'automate linéaire générant S.

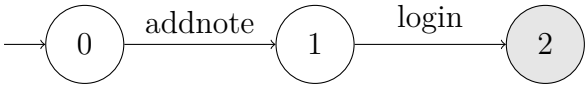

Figure 3.4 L'automate à fusionner

On fusionne alors cet automate dans l'automate père, à la position q.

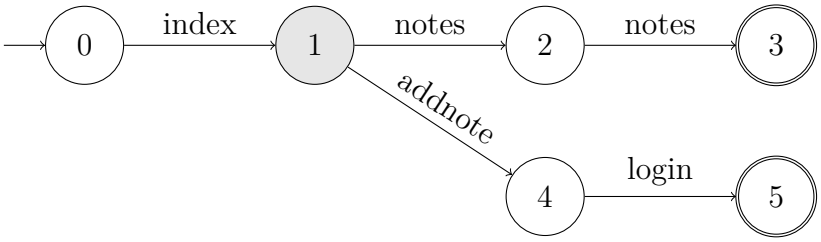

Figure 3.5 Résultat de `kBehavior` sur la seconde trace

Ajoutons maintenant la trace suivante : $S = index.php \rightarrow login.php$. On identifie à nouveau un préfixe dans cette trace et on peut mettre à jour les paramètres en conséquence : $S_{pre} = index.php$, $S = login.php$ et $q = 1$.

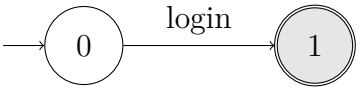

Figure 3.6 L'automate à fusionner

On peut alors fusionner cet automate dans l'automate père.

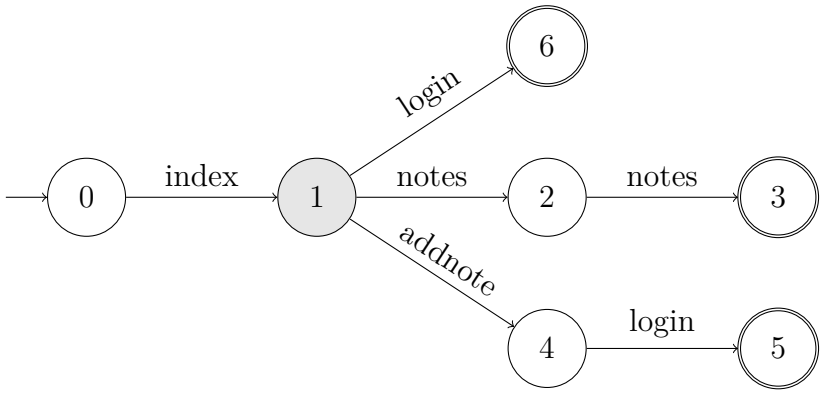

Figure 3.7 Résultat de `kBehavior` sur la troisième trace

La quatrième trace en elle même est un préfixe déjà présent dans l'automate. Les paramètres sont alors $S_{pre} = index.php \rightarrow notes.php \rightarrow notes.php$, $S = \emptyset$ et $q = 3$ et on passe directement à la trace suivante, laissant l'automate inchangé.

Les calculs de la trace $S = index.php \rightarrow index.php$ sont les mêmes que ceux présentés précédemment et on obtient l'automate suivant :

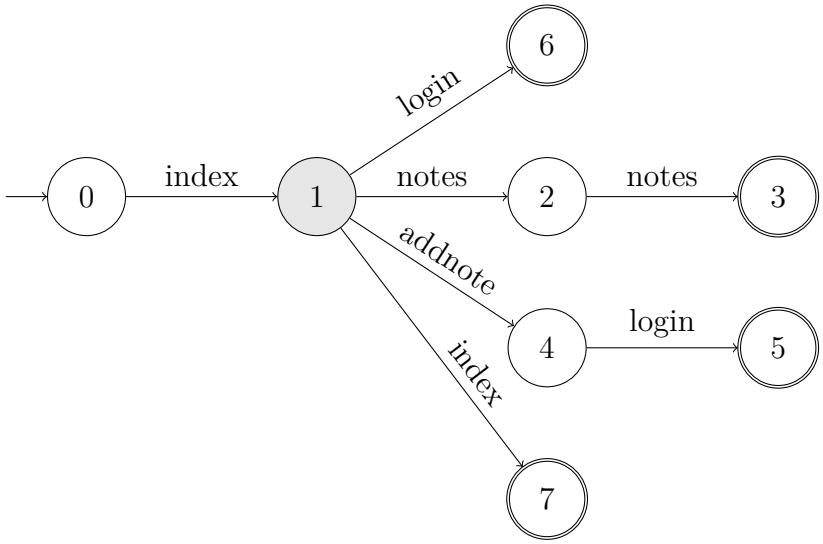

Figure 3.8 Résultat de `kBehavior` sur l'ensemble des traces

3.3.3 La détection d'attaques

Une fois l'automate construit, il faut alors lui fournir des traces d'exécutions dont on ignore la nature. Si l'automate reconnaît le langage, c'est-à-dire si la trace d'exécution correspond à une trace déjà rencontrée et correspondant à une utilisation standard du système, c'est que cette trace correspond elle aussi à une utilisation standard. Dans le cas contraire, si l'automate rejette la trace en question, c'est qu'il n'a jamais rencontré la trace et, si on admet que l'apprentissage a été fait correctement, c'est qu'une attaque vient d'être identifiée.

Travailler avec un automate permet de repérer précisément à quel moment la trace a cessé d'être reconnue, en repérant le dernier état atteint par cette trace. Si cela ne permet pas, dans le cas général, de diagnostiquer l'attaque, une telle information s'avère précieuse car elle donne un indice que peut suivre un analyste cherchant à identifier l'attaque et à corriger le système pour le rendre résistant.

Un autre avantage de l'automate est qu'il peut travailler en temps réel. Lorsqu'il cherche à reconnaître une trace, l'automate n'a besoin que de deux choses : le dernier état atteint par la trace et la nouvelle requête reçue par le serveur. Il n'est donc pas nécessaire d'attendre que la trace soit entièrement générée par le client avant de la fournir à l'automate. Dans cette configuration, ce modèle devient un *dispositif d'alerte*, capable de prévenir un administrateur système qu'une attaque a lieu en temps réel.

3.4 Inférer une fonction de décision

Comme l'a montré (7), un perceptron est un approximateur universel. Nous allons exploiter cette propriété pour approcher la fonction qui, prenant un certain nombre de paramètres que nous avons vu à la partie 3.2.2, détermine si une attaque est en cours ou non.

3.4.1 Exploiter les données

Un réseau de neurones prend un vecteur en entrée, et rend un vecteur en sortie. Contrairement à l'automate, une chaîne de caractères brute est inexploitable par un tel modèle. Il va donc falloir extraire des données des logs. Un log isolé ne contient que peu d'informations pour un réseau de neurones. Au sein d'une trace, il peut prendre beaucoup plus de sens car il contribue à plusieurs paramètres de celle-ci. Prenons, par exemple, la taille de la réponse du serveur. Celle de la réponse est proportionnée à la ressource demandée au serveur. Elle n'a donc, de façon isolée, qu'un intérêt limité. Si on considère la taille totale de la trace, c'est-à-dire la somme des tailles des réponses du serveur, on peut obtenir des informations

supplémentaires : une trace de taille anormalement élevée peut faire redouter qu'un hacker cherche à monopoliser beaucoup de ressources du serveur afin de mener à bien une attaque de type déni de service.

Cependant, devoir attendre que la trace soit complètement générée nous oblige à effectuer un choix décisif : celui d'abandonner toute possibilité d'analyse en temps réel. Dans un tel contexte, il pourrait être intéressant de doubler notre réseau de neurones d'un modèle prédictif qui à l'aide d'un début de trace serait capable de générer le reste. cela permettrait au réseau de neurones de travailler en temps réel sur ces prédictions. Un tel modèle dépasse néanmoins le cadre de notre étude mais donne des perspectives intéressantes.

Partant de notre choix d'analyser les traces comme un tout, nous allons pouvoir en extraire des paramètres pour notre réseau de neurones. Le choix de ceux-ci est bien évidemment lié aux informations que l'on souhaite en tirer. Cela demande une certaine connaissance des attaques que l'on souhaite pouvoir identifier, aussi bien du mécanisme de celles-ci que de leurs conséquences sur le serveur. Les attaques que nous avons sélectionnées sont au nombre de 4 et font partie des attaques les plus répandues : les injections de code, la brute force, les injections SQL et les injections XSS. Nous reviendrons sur les choix de ces attaques et leurs mécanismes dans le prochain chapitre mais il est nécessaire de les évoquer dès à présent pour justifier les paramètres sélectionnés. Ceux-ci sont au nombre de sept :

Le nombre de clics

Le nombre de clics est plus ou moins équivalent au nombre de requêtes. Un tel paramètre permet de savoir combien de pages ont été visitées par le client au court d'une trace. Un client désirant effectuer une action précise aura un nombre de clics plutôt restreint. Un attaquant, dans le cas contraire, cherchera à explorer le site à la recherche d'une vulnérabilité et pourra générer beaucoup de clics. Par ailleurs, certaines attaques, comme les DoS ou les brute-force sont automatisées sous la forme d'un script et permettent donc à un attaquant de générer beaucoup plus de clics qu'un utilisateur humain.

La quantité d'images

Cette information est multiple et peut être à double tranchant. Lorsqu'un client se connecte sur un site web au travers d'un navigateur, celui-ci va automatiquement charger les images de la page sur laquelle il se trouve. Un robot rédigé par un hacker n'est pas forcément intéressé par ces documents et ne va donc pas les télécharger, générant un taux d'images très faible voire nul. Mais d'un autre côté, un robot cherchant à effectuer une attaque de déni de service

va rechercher la page web consommant le plus de ressources et va envoyer énormément de requêtes vers cette page, saturant le serveur. Une fonction d'activation différente serait utile pour modéliser ce genre de comportement en utilisant une tangente hyperbolique à valeurs dans $[-1, 1]$.

Le nombre d'erreurs

Encore une fois, un client qui parcourt le site a peu de chance de générer des erreurs. La majorité des codes HTTP retournés sera donc des codes 200 ou dans cette série. Un hacker pourra se heurter à des ressources non disponibles ou à des privilèges trop faibles, générant des erreurs de la série 400 ou 500.

Les requêtes sans référent

Le référent est une information automatiquement transmise par le navigateur. Elle renseigne l'URL de la page contenant le lien suivi. Un script écrit par un hacker ne renseignera que peu cette information, ou celui-ci peut choisir manuellement de masquer ses traces en ne renseignant pas ce champ.

Les requêtes sans user-agent

Le user-agent est un champ renseigné par le navigateur permettant au serveur d'identifier la version de celui-ci ou le moteur de rendu qu'il utilise. Le choix de considérer ce paramètre pour le réseau de neurone découle des mêmes motivations que pour le référent.

La quantité de données

La somme des poids de toutes les requêtes effectuées permet de mesurer la quantité de données réclamées au serveur. Cela peut permettre de détecter des attaques cherchant à gonfler les flux de données transférées du serveur comme dans le cas du déni de service.

Le nombre de requêtes avec paramètres

Un attaquant effectuant une attaque de force brute ou bien cherchant un formulaire vulnérable va avoir tendance à générer un nombre élevé de requêtes avec des paramètres. Contrôler ce nombre peut aider un modèle à détecter ce genre de comportements.

3.4.2 Des paramètres supplémentaires

Les paramètres que nous avons choisis nous semblent pertinents, en tenant compte des spécificités de notre serveur web. Néanmoins, il en existe d'autres pouvant être utilisés par un perceptron et permettant d'agréger de l'information. En voici quelques-un.

Le nombre de requêtes séquentielles consécutives

Ce paramètre sert à extrapoler le comportement d'un utilisateur du site. Les requêtes séquentielles consécutives sont des requêtes vers des pages toutes situées dans le même répertoire. C'est un paramètre très lié à l'organisation des fichiers sur le serveur mais dans la plupart des cas, les pages web correspondant aux mêmes fonctions sont situées dans des répertoires similaires. Ainsi, durant une session de navigation, les thèmes des pages web consultées par le client étaient corrélés, le nombre de requêtes séquentielles consécutives tend à être élevé. A l'inverse, un robot peut parcourir le site web en largeur et non en profondeur, explorant les liens peu importe le thème de pages web qu'ils permettent d'atteindre. Il génèrera donc un nombre de requêtes séquentielles consécutives assez bas.

L'indice de popularité de page - PPI

Une hypothèse raisonnable est de considérer que les pages webs ne sont pas toutes consultées par les clients avec la même fréquence. Certaines pages sont plus populaires que d'autres. Des utilisateurs normaux vont donc avoir tendance à consulter des pages populaires alors qu'un robot n'aura pas de telles considérations, ou ne sera pas intéressé par le même genre de page. Un inconvénient majeur de cet indice est qu'il est calculé sur l'ensemble des traces. Notre modèle n'est certes pas capable de traiter les données au fur et à mesure qu'elles sont générées mais il peut tout de même analyser une trace dès qu'elle est générée. Le PPI ne peut servir que pour une analyse encore plus tardive. Il se définit ainsi. Pour une page i, le PPI vaut :

$$PPI(i) = -\log(\frac{Nombre\ de\ requêtes\ pour\ la\ page\ i}{Nombre\ total\ de\ requêtes})$$

Une fois le PPI calculé, on peut déterminer le PPI pour une trace S. En notant $R_S(i)$ le nombre de requêtes de la page i durant la trace S, le PPI a pour expression :

$$PPI_{Session}(S) = \frac{\sum_{i=1}^{N}((max(PPI) - PPI(i)) * R_S(i))}{\sum_{i=1}^{N} R_S(i)}$$

3.4.3 La sortie

Déterminer les paramètres d'entrée du réseau est une étape nécessaire mais choisir une sortie l'est tout autant. Nous allons chercher ici à savoir si le réseau subit une attaque ou non mais aussi tenter de qualifier l'attaque. Nous avons identifié quatre attaques. Nous allons donc avoir cinq paramètres de sortie illustrés par cinq neurones. Chaque neurone correspond à la probabilité de se retrouver dans chaque situation : un client normal ou une des quatre attaques.

3.4.4 La topologie du réseau

Une fois les paramètres et la fonction du modèle déterminés, il faut choisir la topologie du perceptron, c'est-à-dire le nombre de couches cachées et le nombre de neurones par couche. Il n'existe pas de formule permettant de déterminer la quantité de neurones et de couches optimales. Il existe pourtant quelques lignes directrices permettant de ne pas tout remettre entre les mains du hasard. Cependant, des ajustements sont généralement nécessaires après les premiers tests, comme par exemple retirer ou ajouter une couche ou un neurone.

Notons tout d'abord qu'une idée répandue est que les perceptrons doivent avoir au moins une couche cachée. En réalité, rien n'empêche un perceptron de ne fonctionner qu'avec une couche d'entrée branchée sur une couche de sortie. Un tel modèle est d'ailleurs très performant pour apprendre des fonctions linéaires. La fonction que nous désirons obtenir n'est pas linéaire et notre modèle devra comprendre au moins une couche cachée.

L'apprentissage fera l'objet de la prochaine partie mais il est déjà important de noter qu'un réseau de neurones complexe mettra plus de temps à apprendre et nécessitera une quantité de données plus importante.

La détermination de la topologie du réseau revient à trouver un équilibre entre deux choses. La première, c'est que le modèle doit être suffisamment complexe pour répondre correctement à la question qu'on lui pose. La seconde est qu'il ne doit pas être trop complexe afin de permettre la généralisation. La généralisation est un concept fondamental de l'apprentissage machine, c'est lui qui se cache derrière le danger du surapprentissage. La généralisation permet à un modèle d'être performant sur des données sur lesquelles il n'a pas appris.

La quantité des données joue également un rôle important dans cet exercice. Un modèle plus complexe aura besoin de plus de données pour ajuster les différents paramètres, le taux d'apprentissage de ceux-ci étant dilué dans la complexité du réseau. Repasser plusieurs fois sur les mêmes données permet, en théorie, d'apprendre sur plus d'informations mais cela réduit également les capacités de généralisation du réseau et ne constitue donc pas une solution.

Un autre principe, assez intuitif, est celui du *fan-in* ou du *fan-out*. Cela correspond à un nombre de couches cachées respectivement inférieur et supérieur à celui de la couche précédente. Lorsque le but du réseau est d'extraire des caractéristiques ou de réduire le nombre de paramètres, le *fan-in* est recommandé, parce qu'il va permettre d'agréer des paramètres d'entrée. Dans notre cas, nous voulons surtout étudier les corrélations entre les différents paramètres et leurs conséquences sur la présence ou non d'une attaque. Nous allons donc avoir plus de couches cachées que de couches d'entrée.

3.4.5 L'apprentissage

Comme nous l'avons vu dans la partie précédente, les neurones contiennent une fonction d'activation f telle que $y = f(\omega_0 + \sum_{i=1}^{n} \omega_i x_i)$, y étant la sortie du neurone, $\{\omega_i\}_{0 \leq i \leq n}$ les paramètres et $\{x_i\}_{0 \leq i \leq n}$ les variables de sortie de la couche précédente. Il existe plusieurs formes de fonctions d'activation étant plus ou moins pertinentes selon les rôles du perceptron. Dans notre cas, nous avons choisi la sigmoïde, c'est-à-dire :

$$f : x \mapsto \frac{1}{1 + e^{-x}}$$

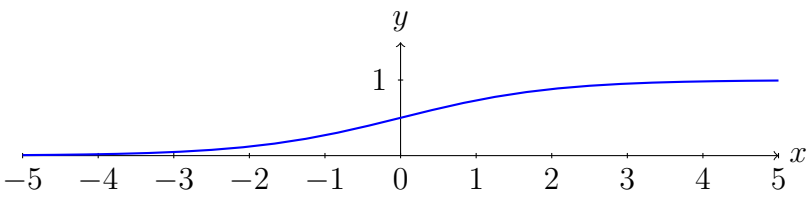

Figure 3.9 La sigmoïde

La sigmoïde est une fonction de $\Re$ dans $[0; 1]$ ce qui en fait une fonction idéale pour analyser des paramètres et s'en servir pour classer l'entrée entre deux catégories.

Une fois la fonction d'activation choisie, il est possible de déterminer les erreurs. Le mécanisme d'apprentissage d'un réseau de neurones est fait d'erreurs et d'ajustements. L'erreur correspond à une différence entre une valeur attendue et la valeur obtenue. Cette erreur permet d'ajuster les poids de chaque neurone pour que ceux-ci répondent mieux (c'est-à-dire avec une erreur plus faible) pour une entrée similaire ultérieure. Encore une fois, il existe plusieurs façons de calculer cette erreur. La façon la plus répandue, et celle employée dans notre modèle, est l'*erreur dérivée* e_i calculée entre la valeur o_i retournée par le neurone i, la valeur désirée d_i, étant donnée une fonction d'activation f. Elle s'exprime différemment pour la couche de sortie et pour les couches cachées. Pour la couche de sortie on obtient :

$$e_i = f'(o_i)(d_i - o_i)$$

On peut ensuite calculer l'erreur d'un neurone i dans une couche cachée en fonction des erreurs e_j de tous les neurones j de la couche suivante et des poids $\{\omega_{i,j}\}_{0 \leq j \leq n}$:

$$e_i = f'(o_i) \sum_{j=1}^{n} e_j \omega_{i,j}$$

Une fois ces erreurs calculées, on peut mettre à jour les poids des différents neurones. Pour cela, on calcule les variations des poids de l'influence de chaque neurone i sur le neurone j $\{\Delta\omega_{i,j}\}_{0 \leq i,j \leq n}$ en fonction de l'erreur dérivée du neurone j, de la valeur de sortie du neurone i et du *taux d'apprentissage* η :

$$\Delta\omega_{i,j} = \eta e_j o_i$$

La détermination du taux d'apprentissage est à nouveau au choix du concepteur de modèle. Il s'agit d'un équilibre entre rapidité et convergence. Un taux d'apprentissage trop faible donnera un réseau de neurones apprenant trop lentement, nécessitant alors beaucoup plus de données et de temps que nécessaire pour converger. À l'inverse, un taux d'apprentissage trop élevé donnera un réseau de neurones ne convergeant pas, oscillant autour de une ou plusieurs positions d'équilibre mais étant incapable de se stabiliser. Faire dépendre le taux d'apprentissage de la taille de l'espace d'apprentissage est généralement conseillé. On obtient donc un taux d'apprentissage de la forme :

$$\eta = \frac{A}{N_{apprentissage}}$$

A est une constante et $N_{apprentissage}$ est la taille de l'espace d'apprentissage. La valeur de A est à déterminer empiriquement et à modifier au fur et à mesure des essais, en surveillant l'évolution de l'erreur moyenne (généralement l'erreur quadratique moyenne, somme des carrés des erreurs des neurones de sortie). Il est conseillé de commencer avec une valeur relativement faible de A et de l'augmenter petit à petit.

Une fois que l'on est capable de mettre à jour les poids, on peut développer un algorithme pour le mécanisme de rétropropagation employé pour notre perceptron est le mécanisme standard. Son fonctionnement est expliqué dans l'algorithme 2.

Algorithm 2 Rétropropagation

```
1: procedure RETROPROGATATION                    ▷ Algorithme de la rétropropagation
2:     Construire le réseau avec le nombre de couches cachées choisi
3:     Initialiser les poids des neurones à des valeurs basses
4:     répéter
5:         Choisir une donnée d'apprentissage
6:         Copier les variables d'entrée dans les neurones de la couche d'entrée
7:         Propager le signal à travers tout le réseau en activant les neurones des couches
   cachées jusqu'à la couche de sortie
8:         Calculer l'erreur dérivée entre les valeurs obtenues dans la couche de sortie et les
   valeurs attendues
9:         Rétropropager la somme des erreurs et des poids sur l'ensemble des couches cachées
10:        Mettre à jour les poids de chaque neurone
11:    jusqu'à Erreur suffisamment basse
12: fin procedure
```

3.5 L'utilisation du réseau

Une fois que le réseau a appris sur les données d'apprentissage, vient le moment de l'utiliser pour déterminer les probabilités d'être dans une situation ou une autre. Pour ce faire, on fixe les valeurs des neurones d'entrée à celles extraites de la trace que l'on considère. On active ensuite les neurones de la deuxième couche, puis ceux de la suivante etc... jusqu'à activer la couche finale. On lit ensuite le résultat du perceptron dans les sorties des neurones de sortie.

Les paramètres de sortie étant à valeur dans $[0; 1]$ (car la fonction d'activation sigmoïde est à valeur dans cet intervalle), ils sont interprétés comme des probabilités. Néanmoins, rien ne garantit qu'ils correspondent

CHAPITRE 4 EXPÉRIMENTATIONS ET RÉSULTATS

4.1 D'où proviennent les données utilisées ?

L'une des premières problématiques qui se pose lorsque l'on parle d'apprentissage machine est l'origine et la nature des données. Sans donnée, le meilleur des modèles ne présente aucun intérêt.

La problématique étant posée, il n'était pas nécessaire que notre étude repose sur des données issues de l'exploitation de systèmes industriels. Les informations comprises dans les logs peuvent être sensibles pour la sécurité du système ou de ses utilisateurs et l'obtention de leur accès n'est jamais aisé. Nous aurions pu nous heurter ici à une première barrière. Par ailleurs, il est difficile de prévoir la complexité d'un tel système. En effet, rappelons que notre étude ne cherche pas à établir un modèle fonctionnant sur des systèmes industriels, mais tente de transposer des techniques d'analyses d'erreurs d'exécution à un contexte d'attaques informatiques. Par conséquent, notre solution n'ayant pas été optimisée pour un tel usage, il est fort probable qu'elle soit peu performante sur ce genre de système. Utiliser des données industrielles représentaient donc un risque significatif que nous avons éliminé en générant nous-mêmes nos données.

Partant de ce choix, il a fallu trouver un moyen de générer des données aussi semblables que possible à celles d'un serveur authentique. Pour ce faire, nous avons utilisé des robots imitant le comportement d'utilisateurs authentiques et d'attaquants. En enregistrant les périodes d'activité des différents robots, il était possible de labelliser les logs de façon automatique et donc de ne pas avoir à les analyser à nouveau pour déterminer s'ils correspondent à une attaque ou non.

4.2 Le serveur Apache

Les logs générés doivent l'être par un serveur particulier. Notre choix s'est porté sur un serveur web et plus particulièrement un serveur Apache dans la dernière version : la version 2.4.20. Pourquoi un serveur web ? C'est un type de serveur particulièrement courant et exposé aux attaques. Il est aujourd'hui difficile de passer une journée sans consulter un site web ou faire appel à un service web sans même que l'on ne le sache. Leur configuration est, en outre, relativement aisée et rapide. Pourquoi Apache particulièrement ? Apache est le serveur web le plus populaire au monde, devant IIS de Microsoft, avec une part de marché comprise entre 35% et 40% selon les études. Notre étude cherchant à être la plus générale possible nous avons

choisi le premier serveur web. Cependant, les logs des autres serveurs web suivent également une nomenclature. Une partie de notre étude serait à refaire pour exploiter les logs de IIS mais la majorité de notre travail est valable pour tous les serveurs web. Au delà des serveurs web, il y a beaucoup trop d'incertitudes pour se prononcer.

Une fois le type de serveur choisi, il faut le configurer. La configuration en elle-même n'est pas très pertinente dans le cadre de notre étude mais précisons tout de même que le format des logs choisi est le format de base pour Apache. Rappelons les informations que nous aurons à notre disposition : l'adresse IP du client, son identité, celle de la personne ayant demandé la requête, la date, la requête, le code de la réponse, la taille de la réponse, le referer et le user-agent.

Le serveur web, pour pouvoir servir ses clients, doit avoir quelque chose à servir ! L'étape suivante a donc été de construire un site web que les clients puissent explorer (afin de générer des logs). L'architecture du site web s'est voulue volontairement minimaliste pour ne pas apporter une complexité de calcul inutile au moment de l'apprentissage. Par ailleurs, ses fonctionnalités ont été pensées afin de pouvoir supporter les quelques attaques mises en œuvre. L'architecture du site est résumée dans la figure 4.1.

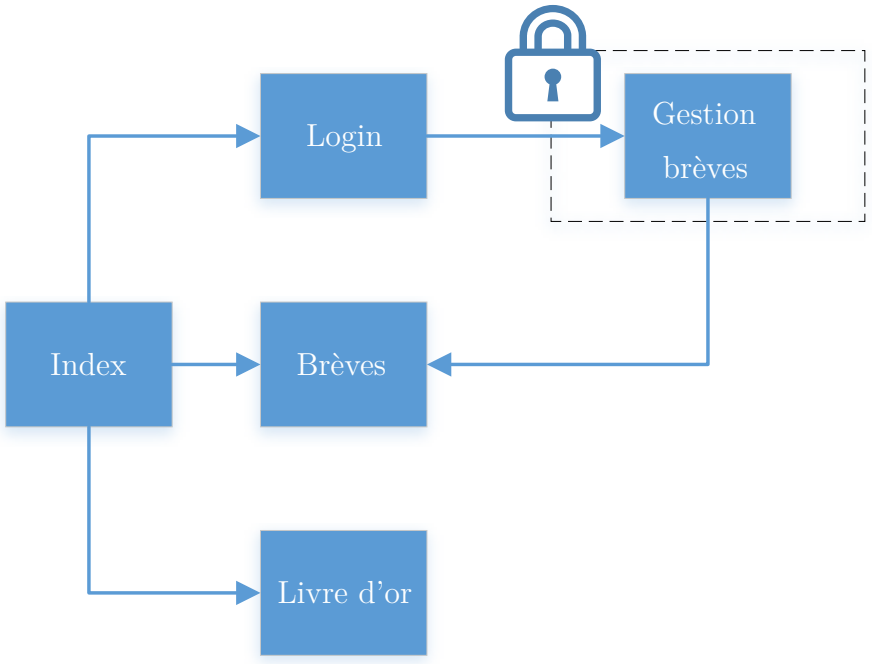

Figure 4.1 Le site web

Comme nous venons de le préciser, les fonctionnalités du site web ont été pensées afin de permettre des attaques. Nous avons donc, comme nous pouvons le constater sur la figure 4.1

un livre d'or, un système de brèves et un espace membre, permettant de gérer les brèves, protégés par un mot de passe. Le contenu géré par le livre d'or et les brèves pourra supporter du contenu multimédia et donc sera une proie idéale pour des attaques DoS. L'espace membre étant accessible via une page de connexion, l'attaque de force brute sera exécutée via ce module. Tout utilisateur étant capable d'insérer du contenu dans le livre d'or, celui-ci sera la cible idéale d'une attaque XSS. Enfin, le formulaire de ce même livre d'or ne sera pas protégé et permettra également d'effectuer des attaques d'injection de code.

Les clients du serveur, c'est-à-dire les robots, ont été placés dans le même réseau que le serveur Apache. Une réflexion a été menée avant de décider de ne pas les répartir en différents sous-réseaux. Il s'est avéré que les modèles choisis n'auraient que peu exploité cette information supplémentaire. Il pourrait être intéressant de faire appel à ce genre d'information (et pas simplement sous la forme d'une *confiance* accordée à un sous-réseau ou à un autre à la façon d'un pare-feu) et cela pourrait être l'objet d'un prolongement de cette étude mais étant donnée la problématique, cette option n'a pas été retenue. Gardons tout de même à l'esprit que l'adresse IP des différents robots est suivie afin de ne pas mélanger les traces générées par les uns et les autres.

4.3 Les attaques

Maintenant que nous avons notre serveur et nos fonctionnalités, nous allons pouvoir expliquer plus en détail les attaques mises en œuvre. Elles sont donc au nombre de quatre : les attaques par force brute, les injections de code, le Cross-Site Scripting et le déni de service.

Les modèles ont été choisis car ils semblaient appropriés aux deux premières attaques que nous allons présenter (déni de service et force brute) et pour lesquelles nous devrions obtenir de bons résultats. En revanche, ils n'ont pas été pensés pour détecter les injections de code et le Cross-Site Scripting et il est donc plus difficile d'anticiper leurs performances. Le réseau de neurones a malgré tout été modifié, dans une certaine mesure, pour prendre en compte des éléments propres à ces deux attaques. Néanmoins, l'objectif reste avant tout de détecter des attaques par force brute et du déni de service.

4.3.1 Déni de service

Une attaque DoS peut prendre plusieurs formes. Le but est néanmoins toujours le même : nuire à la disponibilité du système. Comme son nom l'indique, c'est une attaque qui vise à empêcher le serveur de servir ses clients.

Une attaque DoS commune est le SYN flood. C'est une attaque sur la couche 4, la couche

transport, utilisant le protocole TCP et cherchant à surcharger les ressources du serveur. Afin d'établir une connexion TCP avec un serveur, un client va lui envoyer un segment SYN. Le serveur va réserver des ressources pour la communication et renvoyer un segment SYN-ACK pour accepter la connexion. Le client répond alors par un segment ACK. Une attaque SYN flood consiste, pour le client, à ne jamais envoyer le segment ACK et à bombarder le serveur de segments SYN. De cette façon, le serveur va réserver des ressources pour chaque segment SYN reçu sans les libérer parce que le segment ACK ne sera jamais reçu. Le serveur peut alors consommer toutes ses ressources (espace mémoire, temps de calcul) et n'être plus capable de répondre à des requètes SYN de clients légitimes voire même de planter tout simplement.

Une autre façon de réaliser du déni se service est de surcharger les canaux de communication reliant les clients au serveur. C'est l'attaque que nous allons mettre en œuvre ici. La détection de ce genre d'attaques a d'ailleurs fait l'objet d'un traitement par des réseaux de neurones par (18). Cependant, précisons tout de même que nous n'avons pas mené l'attaque à bien. Nous avons simplement mis en œuvre le protocole d'attaque sans jamais observer de ralentissement. Les raisons de cela sont liées au contexte dans lequel cette expérience a été réalisée. En effet, le client et le serveur correspondaient à deux entités différentes communiquant sur un réseau mais étant hébergées sur la même machine physique. Par conséquent, les canaux de communication avaient un débit bien supérieur à ceux permis par de l'éthernet ou d'autres technologies similaires. Il aurait malgré tout été possible de réduire ce débit mais les logs enregistrant finalement l'*intention* d'attaquer et non le succès de l'attaque en elle-même, il n'y avait que peu d'intérêt à cela.

Afin de réaliser l'attaque, le robot va chercher à consommer un maximum de ressources sur le serveur. Un serveur web est capable de fournir plusieurs types de données : des pages HTML, des images, des vidéos, etc... Certaines données sont plus volumineuses que d'autres. Un robot demandant beaucoup de données volumineuses à un serveur le forcera à consommer sa bande passante et pourra réduire l'espace disponible pour le reste des clients. Il est improbable que le serveur plante suite à une telle attaque mais suffisamment de clients demandant simultanément beaucoup de ressources peuvent sans nul doute provoquer un ralentissement sur le serveur. Une autre conséquence de cela est, dans un environnement Cloud, de forcer le serveur à utiliser plus de ressources et donc d'augmenter sa facture auprès de son fournisseur de Cloud.

Dans notre cas, nous avons écrit un robot recherchant le contenu multimédia le plus lourd hébergé par le serveur et une fois que celui-ci a été identifié, il va bombarder le serveur de requêtes demandant cette ressource. Demander la ressource la plus lourde permet de consommer un maximum de ressources par requête reçue.

4.3.2 Bruteforce

Une authentification sur un site web est souvent réalisée au moyen d'un nom d'utilisateur et d'un mot de passe. Le nom d'utilisateur est généralement publique car il permet aux membres du site de s'authentifier entre eux. Le mot de passe, quant à lui, est privé, connu de son propriétaire uniquement. Il existe plusieurs façons de trouver le mot de passe d'un autre membre et l'attaque par force brute (ou bruteforce) est l'une d'entre elles.

Les attaques par force brute possèdent différents niveaux de sophistication, allant du test de toutes les combinaisons possibles d'un ensemble de caractères données à celui de quelques valeurs. Afin de réduire le nombre de mots de passe possibles, tout en gardant un certain niveau de généralisation, nous avons utilisé un dictionnaire des 10000 mots de passe les plus courants.

La procédure de ce genre d'attaque est assez rudimentaire : l'attaquant essaie de se connecter avec tous les mots de passe de sa liste. Il s'arrête dès que le mot de passe est trouvé. De part le grand nombre de mots de passe à tester, ces attaques sont toujours automatisées.

Il existe des contremesures bien connues permettant de se prémunir contre ce type d'attaque comme les célèbres *captcha* ou bien le verrouillage du module d'identification après plusieurs échecs.

Nous avons installé un espace membre sur notre site permettant aux utilisateurs authentifiés de rédiger des brèves que peuvent lire les clients. Nous allons donc tenter de forcer le module d'authentification. Cependant, l'attaque en elle même s'arrêtant une fois que le mot de passe est trouvé, nous n'allons pas utiliser les identifiants obtenus pour causer davantage de dommages sur le site : le robot s'arrêtera après avoir obtenu confirmation par le site web que le mot de passe est le bon.

4.3.3 Injection de code

Les attaques de type injection de code sont les attaques les plus fréquentes sur les serveurs web d'après l'*OWASP Top 10*[1] en 2010 et 2013. Le but de ces attaques est d'envoyer du code à un interpréteur (SQL, Shell, LDAP, etc.), à travers des commandes ou des requêtes, afin d'exécuter des instructions non désirées. Ce type d'attaque est possible lorsque des points d'entrée dans le système n'ont pas été sécurisés par les administrateurs. Il faut toujours vérifier les données fournies par des utilisateurs. Lorsque cela n'est pas fait, les injections de

1. L'OWASP est une communauté reconnue dans le monde de la sécurité informatique travaillant sur la sécurité des applications web. Tous les trois ans, elle publie un rapport nommé *TOP 10* portant sur les attaques les plus courantes.

code sont possibles.

Dans notre cas, notre site web utilise une base de données SQL et contient un formulaire ayant une vulnérabilité. Nous utiliserons cette vulnérabilité afin d'y insérer une instruction SQL dans laquelle nous demanderons de vider le contenu d'une table.

Une requète SQL a généralement la forme suivante :

```
1   INSERT INTO `relation` (`attribut1`, `attribut2`) VALUES ("valeur1", "
       valeur2");
```

En changeant *valeur2*, on peut facilement insérer une deuxième instruction. On prend donc par exemple pour *valeur2* le texte *"); DELETE FROM `relation`; –* et on obtient :

```
1 INSERT INTO `relation` (`attribut1`, `attribut2`) VALUES ("valeur1", "");
     DELETE FROM `relation`; -- ");
```

L'interpréteur SQL va donc envoyer la première instruction qui génèrera l'insertion d'une nouvelle entrée dans la table *relation* avec la valeur *valeur1* pour l'attribut *attribut1* et une valeur vide pour l'attribut *attribut2*. Ensuite, l'interpréteur exécutera la deuxième instruction qui supprimera toutes les données de la table *relation*. La partie après les deux tirets – correspond à des commentaires et ne sera pas interprétée.

Un mécanisme de protection contre ce genre d'attaque consiste à contrôler la présence de caractères spéciaux et de les traiter correctement afin qu'ils ne puissent pas permettre de générer de nouvelles commandes que l'interpréteur traiterait.

Dans notre situation, le robot effectuant l'injection de code se rend jusqu'à la page web contenant le formulaire vulnérable et effectue l'injection. La commande injectée réalise la suppression de tous les messages contenus dans le livre d'or.

4.3.4 Cross-Site Scripting

Le *Cross-Site Scripting*, en abrégé *XSS*, est une attaque similaire à l'injection de code. A vrai dire, on cherche encore une fois à injecter du code sur le site web mais les leviers utilisés sont cependant différents. L'injection de code telle que nous l'avons vu plus haut consiste à envoyer du code au serveur que celui-ci exécutera via un interpréteur. Dans le cas du XSS, c'est au contraire le client qui exécutera le code. La plupart des navigateurs web activent le JavaScript par défaut ce qui en fait un choix populaire parmi les hackers. Néanmoins, n'importe quel langage exécuté chez le client, comme le Flash ou même le HTML5, est un potentiel vecteur d'attaques. Étant exécuté chez le client, le code injecté permet de récupérer des informations le concernant et permet donc le vol de session, par exemple. Les attaques

XSS sont les troisièmes attaques les plus répandues d'après le TOP 10 de l'OWASP pour les années 2010 et 2013.

La vulnérabilité responsable de l'attaque est généralement la même que pour une injection de code : une entrée non protégée dans un formulaire. Dans notre site, c'est à nouveau une entrée du livre d'or qui sera utilisée pour mener à bien l'attaque. Le langage que nous allons utiliser est du JavaScript et nous allons simplement lever une alerte :

```
1 <script>alert('bonjour')</script>
```

Le code que nous envoyons sur le site web compte donc comme une entrée du livre d'or et sera téléchargé par le client lorsqu'il cherchera à afficher le contenu de celui-ci. Le code téléchargé est interprété par le navigateur du client comme étant du code HTML lui-même contenant du JavaScript. A vrai dire, le client est incapable de détecter du code apocryphe et exécutera toujours le code HTML qu'il recevra.

Nous allons désormais nous intéresser en détail à l'ensemble du protocole expérimental nous permettant d'automatiser la quasi intégralité du processus d'analyse.

4.4 Le protocole expérimental

Les expériences ont été réalisées en trois grandes étapes. La première est la génération des données. Afin que nos modèles apprennent il est nécessaire de leur fournir et donc de générer des données. L'automate n'a uniquement besoin que de traces générées par des clients qui peuvent donc potentiellement être générées par un être humain. Le réseau de neurones en revanche nécessite une quantité de données bien supérieure nécessitant, dans notre cas, l'utilisation de robots. La deuxième étape est le formatage des données. Une fois générés, les logs bruts doivent être raffinés afin d'en extraire les paramètres utilisés par les deux modèles. Enfin, dans la dernière étape, on réalise l'apprentissage à proprement parlé et les tests afin de valider ou non nos hypothèses.

Toutes ces phases ont nécessité l'écriture de programmes informatique. Certaines d'entre elles nécessitant un niveau d'abstraction élevé, un langage objet a été privilégié. Le langage choisi a donc été le C#, permettant également d'exploiter le confort fourni par le framework .NET de Microsoft. Cependant, les calculs effectués par le réseau de neurones étant massivement parallèles, l'utilisation d'un langage plus dédié au calcul parallèle, ou même l'utilisation de processeurs graphiques, serait recommandée.

4.4.1 Générer des données

La première étape consiste à générer des données. Le danger de données artificielles est de les rendre adaptées, volontairement ou non, au système. Il est donc impératif de demeurer le plus neutre possible et de générer des données fidèles à la réalité. Pour se faire, plusieurs pistes ont été suivies comme générer des données manuellement en menant soi-même à bien les attaques et analyser les logs afin que les robots reproduisent le comportement d'un être humain ou introduire une part de hasard. Cette dernière mesure présente également l'avantage de ne pas générer les mêmes logs (ce qui présenterait alors peu d'intérêt).

Afin de respecter certaines conventions suivies par les navigateurs web comme Mozilla Firefox ou Microsoft Edge, les robots clients miment leur comportement, notamment dans la gestion des entêtes avec le référent ou le user-agent. A l'opposé, des robots attaquants fonctionnant de manière automatisée ne s'embarrassent que rarement de tels raffinements et les champs ont donc été laissés vides dans ces cas là.

Le traitement des pages web est réalisé par deux classes : la classe `WebClient` et la classe `HtmlDocument`. Cette dernière est présente dans assembly externe réalisée dans le cadre du projet `Html Agility Pack`. Le projet `Html Agility Pack` fournit un ensemble d'outils pour analyser rapidement et efficacement des pages web afin d'en extraire des balises précises, leurs attributs et les valeurs de ces attributs. La classe `WebClient`, quant à elle, fournit tous les outils pour envoyer des requêtes à un serveur web et en recevoir les réponses.

Après avoir généré du trafic sur le serveur web, et donc des logs, les robots vont récupérer les logs correspondant à leur activité et stocker ces logs dans un fichier. Afin de simplifier l'analyse ultérieure, les noms de ces fichiers permettent d'identifier immédiatement la nature du robot les ayant générés, à savoir : si c'est un robot malicieux ou un robot client et dans le cas d'un robot malicieux, quelle attaque il a effectuée.

4.4.2 Raffiner les logs

Une fois les logs générés, ils doivent être analysés afin d'en extraire les informations identifiées comme utiles. La procédure d'extraction est la suivante. On ouvre un fichier de logs généré par les robots et on le lit ligne par ligne. Les données de chaque ligne sont alors extraites via une expression régulière et sont stockées temporairement. Lorsqu'une fin de trace est identifiée (dans notre cas une requête à la page *index.php*), on enregistre ses données et on reprend l'analyse du fichier de logs. L'enregistrement des données extraites est réalisé dans des fichiers externes afin de ne pas réaliser l'opération plusieurs fois car celle-ci est assez coûteuse en temps.

L'expression régulière réalisant l'extraction des logs est la suivante :

```
(.+)\s(.+)\s(.+)\s\[(.+)\]\s\p{P}(.+)\p{P}\s(\d+)\s(\d+)\s\p{P}(.+)\p{P}\s
   \p{P}(.+)\p{P}
```

L'implémentation des expressions régulières reste propre à chaque langage. Le principe reste le même et la majorité du fonctionnement des expressions régulières reste identique mais certaines spécificités ou typologies peuvent être différentes d'un langage à l'autre. Le but ici est d'exploiter la nomenclature définie par les fichiers de configuration d'un serveur HTTP Apache afin d'extraire les données recherchées. Les données extraites sont les groupes identifiés par des parenthèses `(.+)` ou `(\d+)`, séparées par des espaces `\s`, des crochets `\[ \]` ou des caractères de ponctuation `\p{P}`.

Contrairement à nos prévisions, cette opération a été, de loin, la plus coûteuse en temps. Cela est certainement dû à une expression régulière peu performante. Il serait donc nécessaire d'optimiser cette étape dans le cas d'une utilisation industrielle de ce processus. Néanmoins, peu importe les données recherchées, cette extraction des logs n'est que liée à un serveur HTTP Apache : si l'on recherchait des informations différentes dans les logs d'Apache, cette étape d'extraction serait réutilisable en l'état.

Une fois les données brutes identifiées elles sont analysées et des calculs appropriés sont effectués afin de préparer les entrées des deux modèles suivants. Les fichiers résultant de cette opération ont la forme du tableau 4.1 pour l'automate, et du tableau 4.2 pour le réseau de neurones.

Tableau 4.1 Des entrées pour l'automate

GET /index.php HTTP/1.1
GET /livreor.php HTTP/1.1 HTTP/1.1
GET /livreorsubmit.php HTTP/1.1 HTTP/1.1
GET /livreorsubmit.php HTTP/1.1

Tableau 4.2 Des entrées pour le réseau de neurones

4
0
0,25
1
1
2955
0

Notons que dans le tableau correspondant au réseau de neurones, les données correspondent aux paramètres identifiés au chapitre précédent, à savoir dans l'ordre : le nombre de clics, le taux d'images, le taux d'erreurs, le taux de requêtes avec référent, le taux de requêtes avec user-agent, la quantité de données générées et le taux de requêtes avec paramètres.

Par ailleurs, chaque fichier généré permet d'identifier rapidement le type de robot ayant généré les logs. Chaque robot signe d'une lettre différente le nom des fichiers qu'il génère, en plus d'utiliser l'horodatage pour éviter tout doublon.

4.4.3 L'apprentissage des modèles

La dernière étape est consacrée à l'apprentissage des modèles. Chaque modèle va lire les fichiers générés par le raffinage des logs effectué à l'étape précédente afin de régénérer les données.

L'automate

Dans le cas de l'automate, les seuls fichiers de logs servant à l'apprentissage sont les fichiers générés par les robots clients. L'algorithme utilisé est l'algorithme `kBehavior` décrit au 3.3.2. Les tests quant à eux sont effectués sur l'ensemble des données présentes.

Les tests consistent à fournir une trace à l'automate et lui demander s'il la reconnaît ou pas. Sa réponse est alors enregistrée et comparée à la nature de la trace, c'est-à-dire si le log correspond à une attaque ou pas. La justesse des réponses est alors enregistrée, attaque par attaque, afin d'attester des capacités de l'automate à les détecter.

Le réseau de neurones

Un réseau de neurones peut apprendre de plusieurs façons. Cela est automatiquement réalisé à partir du moment où l'on ajuste les paramètres du modèle. La méthode présentée au 3.4.5

est l'apprentissage *online* où l'ajustement des paramètres est effectué après chaque itération. L'ajustement de paramètres peut être gourmand en ressources. Afin d'éviter ce phénomène, le *traitement par lots* (en anglais *batch*) est souvent employé. Dans ce type d'apprentissage, les données d'apprentissage sont regroupées par lot et les poids ne sont mis à jour qu'après le traitement de chaque lot (l'erreur est cependant calculée après chaque donnée). Les lots doivent, cependant être équilibrés pour les classes au risque de voir le réseau de neurones privilégié une classe plutôt qu'une autre.

Une fois l'apprentissage réalisé, les données du test sont passées au réseau afin de contrôler ses performances. De façon similaire à l'automate, les résultats seront enregistrés afin d'obtenir des statistiques concernant les capacités du modèle à diagnostiquer les attaques.

4.5 Les résultats

Comme indiqué précédemment, la génération des résultats a été effectuée en trois étapes : la génération des logs, leur raffinage et l'apprentissage des modèles.

Commençons par préciser que toutes les opérations ont été réalisées sur le même ordinateur, utilisant un processeur Intel® Core™ i7-47121HQ possédant huit cœurs cadencés à 2,30 GHz et 16 Go de mémoire vive. Les programmes n'ont jamais dépassé le gigaoctet de mémoire utilisée (donc n'ont jamais été ralentis par des accès disque) et n'ont pas bénéficié d'une architecture supportant le multi-threading (n'utilisant donc qu'un seul cœur à la fois). Nous allons voir que beaucoup de fichiers sont utilisés donc les accès disques étant fréquents, ajoutons que le seul disque utilisé a été un SSD, rendant la lecture de ces fichiers plus rapide que pour un disque dur.

Les logs bruts ont été générés par 30 200 exécutions de robots pour un total de 1,81 Go de données. Cette génération a duré environ six heures. Notons tout de même la quantité importante de données générées malgré la simplicité de nos robots, la taille réduite du serveur web et le fait que la source était réduite à un seul client. Ramené à une échelle industrielle, on comprend facilement la nécessité d'outils adaptés pour analyser toutes ces données.

Le raffinage a ensuite été l'étape la plus longue, et de très loin, à cause d'un algorithme d'extraction de données peu optimisé. L'étape a duré environ trente-et-une heures et a généré 400 Mo de données réparties en 30 200 fichiers pour l'automate et 11,7 Mo répartis sur 411 316 fichiers pour le réseau de neurones. Le nombre important de fichiers pour le perceptron provient du fait que certains fichiers de logs contiennent plusieurs traces et que chaque fichier de données du réseau de neurones correspond à une seule trace. Par ailleurs, le fait que ces fichiers ne stockent que les paramètres d'entrée du perceptron (donc sept valeurs numériques)

est responsable de leur faible poids, à l'inverse de ceux de l'automate qui contiennent encore chaque requête auxquelles le serveur a répondues.

La partie qui nous intéresse le plus est évidemment celle de l'apprentissage et des performances des deux modèles. Nous allons la détailler dans les paragraphes qui suivent.

4.5.1 L'automate

Commençons par rappeler que l'automate apprend uniquement sur les traces correspondant à une exécution normale. Cela représente 319 407 traces d'apprentissage. Une conséquence directe d'un si grand nombre de traces couplées à un site web relativement simple est la forte redondance des traces générées. L'extraction des traces des fichiers de données et leur stockage en mémoire prend entre huit et neuf secondes. L'algorithme `kBehavior` est visiblement très performant pour apprendre des traces redondantes car il ne prend qu'environ 1,5 secondes pour traiter l'ensemble des traces.

Nuançons tout de même ce temps de calcul en apparence très court par le fait que la plupart des traces fournies à l'algorithme sont d'une dimension faible. Des traces plus longues, résultat d'un choix de découpage différent de la part de l'administrateur, auraient assurément des temps de calculs plus élevés. Par ailleurs, un site web plus complexe génèrerait des arités supérieures pour les états de l'automate et l'exploration du graphe s'en retrouverait allongée. Néanmoins, la complexité de l'algorithme est due à deux calculs : la recherche de préfixes et la recherche de sous-séquences. La recherche de sous-séquences est bien plus coûteuse en temps mais elle intervient de moins en moins au fur et à mesure que l'automate se construit (car les préfixes sont de plus en plus longs). Une trace redondante apparaitra comme un préfixe dont le temps d'identification sera faible (ne dépendant que de l'arité des états de l'automate). Par conséquence, même si ses performances seraient moins impressionnantes, l'algorithme `kBehavior` resterait très certainement performant pour traiter des ensembles fortement redondants.

L'automate obtenu par l'exécution de `kBehavior` sur les 318 407 traces est un automate possédant dix états et neuf transitions. Il a été fidèlement représenté sur la figure 4.2.

Après sa génération, le programme de test va envoyer toutes les traces à l'automate qui va alors dire s'il les reconnaît ou pas. Afin d'avoir des statistiques fiables, nous gardons la trace des traces reconnues et des traces non reconnues. De cette manière, nous pouvons comparer les performances du modèle pour chaque type d'attaques. Elles sont sans équivoques et sont résumées dans le tableau 4.3.

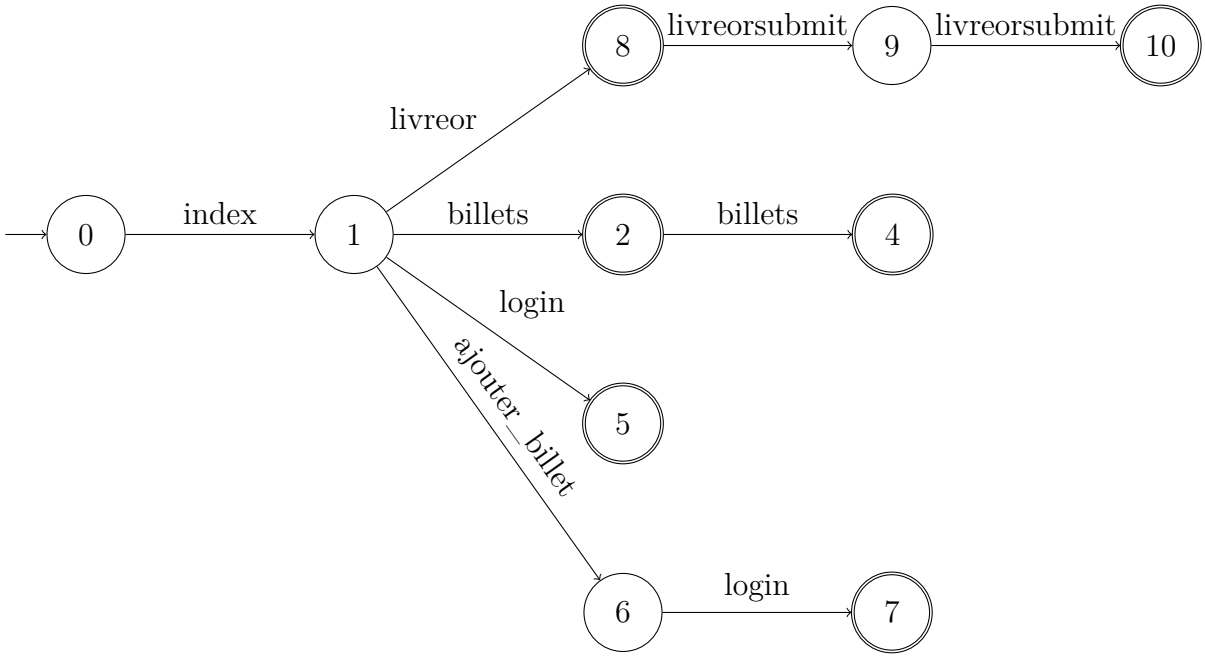

Figure 4.2 Résultat de `kBehavior` sur l'ensemble des traces

Tableau 4.3 Résultats de la reconnaissance des traces pas l'automate

Type de trace	Traces utilisées	Traces reconnues	Taux de succès
Normale	310 191	310 191	1
Injection SQL	5 891	5 891	1
Bruteforce	2 594	0	0
Déni de service	2 526	0	0
XSS	3 329	3 329	1

Commençons par noter que l'automate reconnaît l'intégralité des traces que `kBehavior` a utilisées pour réaliser l'apprentissage. C'est rassurant et cela permet de valider les fonctionnalités de `kBehavior`. Par ailleurs, l'automate est capable de reconnaître des attaques par force brute et des attaques de déni de service de façon certaine. Néanmoins, détecter des attaques d'injection de code ou de XSS est impossible. Cela était prévisible et s'explique assez simplement.

Dans les attaques par force brute et de déni de service, l'attaquant exploite l'architecture du site web (multiples appels à une quantité très restreinte de ressources). Les empreintes laissées par l'attaquant rendent donc compte de ce caractère. Au contraire, dans le cas du XSS et de l'injection, le vecteur d'attaque se présente sous la forme de contenu passé en paramètres dans une requête. L'automate est incapable d'observer les paramètres et ne peut

donc pas faire la distinction entre une attaque et un comportement normal. Par ailleurs, les attaques XSS ou d'injection SQL sont rarement perpétrées par des robots et le comportement de l'attaquant, au sens des ressources consommées, sera le même que celui d'un client.

Les temps de calculs de l'automate pour cette phase sont liés à notre implémentation de l'automate et du parcours au sein de celui-ci mais précisons tout de même le temps de calcul de l'identification des traces : environ 1,5 secondes pour les 324 531 traces.

Nous reviendrons plus tard sur les perspectives de l'utilisation d'un automate mais notons tout de même dès à présent que nous avons ici un modèle très robuste sur certaines attaques apprenant très rapidement et donnant des résultats de façon quasi immédiate.

4.5.2 Le réseau de neurones

En préambule de cette section concernant le réseau de neurones rappelons que, contrairement à l'automate qui se construit de manière incrémentale, le réseau de neurone converge vers une solution. Cette convergence se mesure à la variation de l'erreur au fur et à mesure de l'apprentissage. Une fois que cette variation est inférieure à un certain seuil déterminé arbitrairement, on dit que le modèle a convergé. Nous allons voir que plusieurs modèles seront appris avec des complexités différentes. Une complexité supérieurs demandera plus de données pour réaliser l'apprentissage car la convergence sera moins rapide. Tous les résultats présentés concernent des modèles ayant effectivement convergé.

Commençons par mentionner les données utilisées. L'apprentissage par lot que nous avons employé ici demande des classes de données équilibrées. Nous avons donc été limité par la classe la moins représentée pour un total de 2 526 traces par classe. La première opération effectué à l'initialisation du programme réalisant l'apprentissage est la régénération des traces à partir des fichiers résultant de la précédente opération de raffinage. Cette régénération prend environ 5 secondes. La phase suivante est l'apprentissage en lui-même du réseau en fonction de la topologie choisie. La durée de la phase d'apprentissage et la phase de test est fonction de la complexité de modèle mais notons qu'elles n'ont jamais dépassé la minute de calcul pour tous nos tests. Les temps du calcul étaient donc très raisonnables.

Une fois que nous possédons les données, il nous faut déterminer quelle sera la topologie du réseau, et donc sa complexité. Nous avons vu au chapitre précédent qu'un certain nombre de paramètres devraient être pris en compte dans la détermination du réseau. Nous avons décidé d'utiliser différentes topologies afin de vérifier cela mais notre hypothèse était qu'il y aurait de meilleurs résultats avec une couche cachée de dix neurones, réalisant le fade-out présenté à la section 3.4.4. Nous avons donc sélectionné six topologies différentes : un modèle

linéaire sans couche cachée, quatre modèles avec une seule couche cachée de respectivement cinq, dix, vingt et cinquante neurones et un dernier modèle avec deux couches cachées de vingt neurones chacune.

Afin de diluer l'influence de l'initialisation des poids dans les neurones, l'apprentissage a été réalisé sur vingt modèles différents pour chaque topologie. Les bilans ont ensuite été déterminés sur la moyenne des résultats individuels de chaque modèle.

La performance de ces modèles a été évaluée en utilisant deux indicateurs. Le premier est le taux d'erreur du modèle. Un taux de 0 indique que toutes les attaques ont correctement été déterminées et qu'aucun faux positif n'a été remonté. Un taux de 1 indique que le perceptron a été incapable de déterminer correctement la moindre classe. L'autre indice a été l'écart-type. En effet, il est possible que le réseau de neurone privilégie une classe par rapport à une autre : qu'il ne détecte jamais une attaque, résultant en un taux d'erreur relativement bas. Ce genre de situation sera illustrée par un écart-type élevé. Enfin, les espaces de test étaient équilibrés avec environ 500 représentants pour chaque classe. Les résultats sont exposés dans le tableau 4.4.

Tableau 4.4 Résultats des tests sur le perceptron

Topologie	**Taux d'erreur moyen**	**Écart-type moyen**
Linéaire	0,753	597
5	0,757	792
10	0,720	705
20	0,734	620
50	0,655	631
20, 20	0,674	697

Les résultats du réseau de neurone sont très mauvais. Non seulement le taux d'erreurs est très élevé mais l'écart-type l'est également, ce qui tend à montrer que le réseau de neurones privilégie massivement certaines classes. Une étude plus approfondie des résultats montre que c'est le cas. Malgré un espace d'apprentissage équilibré, le réseau de neurones n'attribue généralement aucune trace à une classe et privilégie au contraire une ou deux classes recevant la majorité des traces. Notons qu'il ne semble pas y avoir de classe particulièrement privilégiée : deux réseaux apprenant sur les mêmes données ne pencheront pas vers les mêmes classes. Cela est dû à l'initialisation des poids dans les neurones qui se fait de façon aléatoire.

Étant donné les faibles résultats du réseau de neurones, nous avons décidé de simplifier le modèle. Le perceptron ne doit donc plus chercher à déterminer la classe de la trace soumise parmi cinq mais simplement de détecter si elle correspond à un certain type d'attaque ou

non. La réponse du modèle n'était donc plus un vecteur comportant cinq estimations mais une seule : une attaque est-elle en cours ou non.

Les données fournies à l'automate ont été préalablement découpées en listes de tailles égales : une liste de traces correspondant à des comportements normaux, une autre liste pour les attaques XSS, une autre pour les attaques par force brute, etc. De cette façon, lorsque l'on a choisi d'entrainer un réseau de neurones à reconnaître des attaques XSS, nous avons fusionné les deux listes de traces normales et d'attaques XSS afin de les passer au perceptron.

Les expériences ont été réalisées dans le but d'étudier l'influence des couches cachées sur le modèle, ainsi que la capacité du perceptron à reconnaître une attaque plutôt qu'une autre. Pour ce faire, chaque attaque a été apprise 20 fois par un réseau de neurones (afin, encore une fois, de diluer des résultats singuliers). Les réseaux de neurones avaient également une complexité de plus en plus élevée : aucune couche cachée, une couche cachée avec 5, 10, 20 puis 50 neurones et enfin deux couches cachées de 20 neurones. L'erreur d'un modèle correspondait à son taux d'erreur sur l'espace de test. Afin de juger de la précision et de la fiabilité des modèles, nous avons calculé la moyenne des erreurs (jugeant de la performance du modèle) et leur écart-type (évaluant la constance dans la convergence des modèles). Les résultats ont été résumés dans le tableau 4.5 et les figures 4.5, 4.3, 4.4 et 4.6.

La première chose que l'on remarque sur le tableau 4.5 est que les erreurs sont toujours élevées, bien que plus faibles que dans notre premier modèle, et que les écarts-types sont très inégaux selon les attaques. Globalement, les écarts-types restent assez élevé, témoignant d'une difficulté du modèle à converger vers une même solution. Cela veut dire que les valeurs initiales des poids des neurones du réseau, déterminées aléatoirement, ont une grande influence sur la solution vers laquelle converge notre perceptron. C'est le cas lorsque la fonction objectif possède beaucoup de minimums locaux et que l'on a tendance à converger vers un minimum différent à chaque apprentissage. La fonction objectif étant liée aux paramètres, le choix de ceux-ci est certainement à remettre en question.

Discutons maintenant des influences des couches cachées. Afin de visualiser plus clairement l'incidence des couches cachées, nous avons reproduit les résultats du tableau 4.5 dans quatre histogrammes représentant chacun une attaque (figures 4.5, 4.3, 4.4 et 4.6). La complexité 1 représente le cas linéaire, la complexité 2 le cas où une seule couche cachée de cinq neurones est présente et ainsi de suite jusqu'à la complexité 6 représentant deux couches cachées de vingt neurones chacune.

Il ressort de ces données que les injections SQL et le déni de service ne profitent absolument pas d'une augmentation de complexité. Au contraire, les performances des deux modèles sont meilleures dans le cas linéaire. Cela traduit généralement une situation de surapprentissage.

Tableau 4.5 Résultats des tests sur les modèles simplifiés

Type d'attaque	Topologie	Erreur moyenne	Écart-type
Bruteforce	Linéaire	0,256	0,185
Bruteforce	5	0,487	0,258
Bruteforce	10	0,362	0,244
Bruteforce	20	0,253	0,215
Bruteforce	50	0,233	0,197
Bruteforce	20, 20	0,364	0,313
DoS	Linéaire	0,416	0,155
DoS	5	0,535	0,156
DoS	10	0,447	0,133
DoS	20	0,402	0,159
DoS	50	0,407	0,158
DoS	20, 20	0,430	0,154
SQL	Linéaire	0,380	0,236
SQL	5	0,432	0,219
SQL	10	0,435	0,279
SQL	20	0,453	0,283
SQL	50	0,461	0,316
SQL	20, 20	0,495	0,235
XSS	Linéaire	0,396	0,320
XSS	5	0,456	0,207
XSS	10	0,439	0,286
XSS	20	0,447	0,298
XSS	50	0,344	0,286
XSS	20, 20	0,354	0,228

Cependant, dans notre cas, il est plus probable que l'explication soit ailleurs, avec des paramètres qui ne sont certainement pas adaptés au problème, car les cas de surapprentissage ne se manifeste que rarement aussi rapidement, et rien ne laisse supposer que la fonction objectif soit linéaire.

Dans le cas des attaques par force brute et XSS, le modèle semble plus performant, notamment dans le cas de la force brute avec un écart-type et un taux d'erreur significativement plus faibles que les pour autres. Dans le cas de l'attaque XSS, les modèles ont tendance à bénéficier d'une augmentation de complexité. Ainsi, même si les performances ne sont pas satisfaisantes pour une utilisation en situation réelle, il est manifestement possible d'améliorer les modèles, non pas simplement en augmentant la complexité de ceux-ci à l'infini mais en étudiant également les paramètres d'entrée du réseau de neurones afin d'obtenir une fonction

objectif dont le minimum global serait plus facilement atteignable.

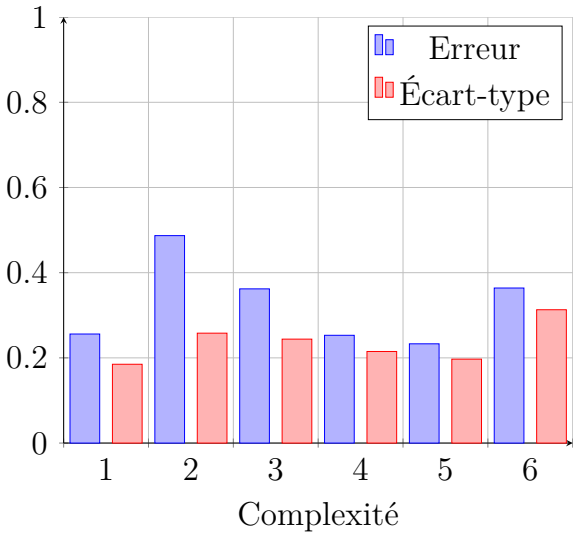

Figure 4.3 Attaque par force brute

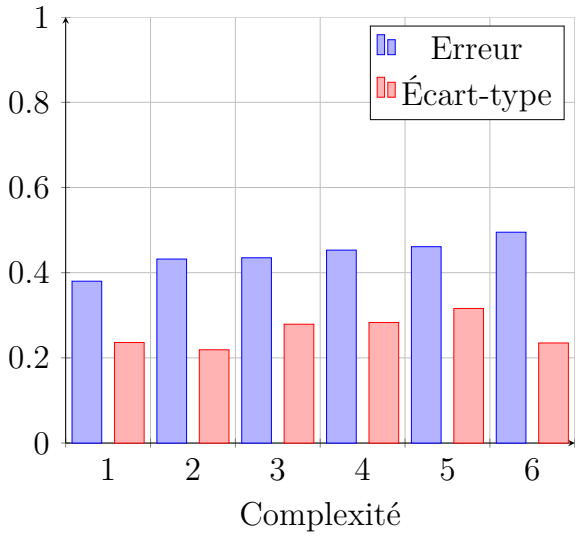

Figure 4.4 Déni de service

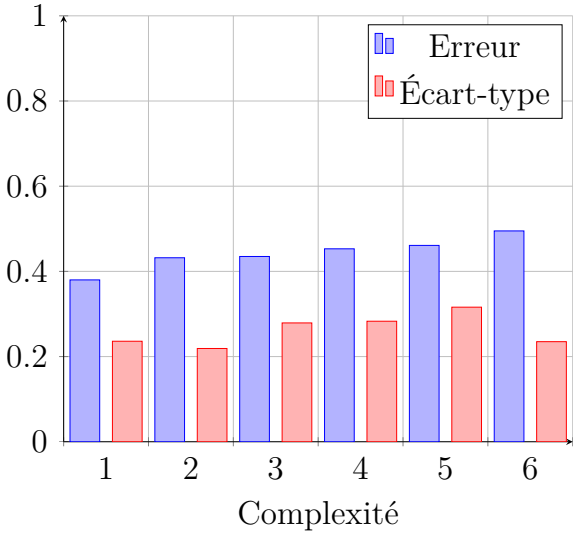

Figure 4.5 Injection SQL

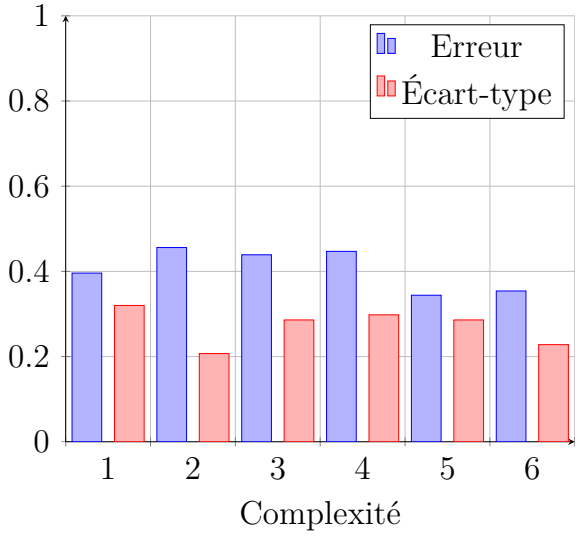

Figure 4.6 Attaque XSS

CHAPITRE 5 CONCLUSION

Les résultats étant présentés, nous allons pouvoir conclure ce projet. Pour ce faire, nous allons dans un premier temps présenter une synthèse du travail, avant d'exprimer les limites des modèles. Enfin, nous terminerons en proposant des axes de recherche afin d'approfondir nos travaux.

5.1 Synthèse des travaux

Nous avons présenté plusieurs étapes. Tout d'abord, et c'était le but du mémoire, nous avons détaillé deux modèles permettant d'analyser le comportement des utilisateurs d'un site web et de détecter des attaques. Le matériau de base de ces modèles est le log, ligne de texte généré par le serveur web. En ce sens, les deux modèles sont non intrusifs et ne nécessitent donc pas de modification du système pour fonctionner.

Dans un premier temps, les logs ont été analysés afin de faciliter la récupération des données. En effet, beaucoup de méthodes d'extraction de données ont été développées dans la littérature mais elles se fondent généralement sur le fait que les logs sont de simples chaînes de caractères sans structure particulière. Les logs de serveurs web suivent, au contraire, une nomenclature bien précise, et l'exploitation de ces informations permet de récupérer plus de paramètres et ce de manière plus sûre. Bien évidemment, une fois ces paramètres extraits il faut les exploiter. C'est le but des deux modèles que nous avons développés.

Le premier modèle est un automate : un modèle déterministe fonctionnant directement sur les requêtes reçues par le serveur et utilisant le parcours des clients sur le site web pour inférer sur leur comportement. Un automate est un modèle binaire : il reconnaît ou ne reconnaît pas une trace mais n'est pas capable de la qualifier. L'apprentissage de l'automate a été réalisé au moyen de l'algorithme kBehavior, développé par (11). Les performances de l'automate sont très contrastées, soit authentifiant à coup sûr une attaque, soit au contraire ne pouvant pas la détecter. L'apprentissage d'un tel modèle est très rapide et la vitesse est plus liée à l'hétérogénéité des données qu'à leur quantité.

Le second modèle est un réseau de neurones, un perceptron multicouches plus exactement. Son rôle consiste, à partir de paramètres extraits d'une trace, de déterminer si ladite trace correspond à une utilisation normale ou à une attaque et dans le cas où une attaque est repérée, de l'identifier. La détermination des paramètres d'entrée du réseau de neurones a fait l'objet d'une réflexion approfondie, alimentée par les spécificités des attaques que l'on a

cherchées à identifier.

Ces deux modèles s'inscrivent au sein d'un protocole expérimental que nous avons développé afin d'automatiser la génération des données. Ce protocole a nécessité la rédaction de robots générant du trafic sur le serveur, légitime ou fallacieux. Les logs générés par les robots étaient ensuite récupérés, raffinés et enfin fournis aux deux modèles. Puis les résultats retournés par les deux modèles ont été analysés pour déterminer leurs capacités à détecter des attaques.

5.2 Les limites de la solution proposée

Bien que les résultats de l'automate correspondent à nos attentes, ceux du réseau de neurones sont assez frustrants. Le perceptron est tout de même légèrement plus performant pour l'une des attaques que l'on cherchait spécifiquement à analyser. Cependant, on ne peut nier que les résultats ne vont pas dans le sens d'une utilisation telle quelle du modèle pour la détection des attaques. Néanmoins, ils sont encourageants car ils ouvrent la voie à d'autres modèles fondés sur les réseaux de neurones comme nous allons le voir dans le paragraphe suivant.

L'automate, quant à lui, est parfaitement capable de détecter les deux attaques prévues. Il ne peut néanmoins pas identifier les injections de code et le Cross-Site Scripting. C'est un modèle finalement très spécialisé n'ayant pas la capacité de généraliser si les symptômes des attaques sont trop différents. C'est une limite qu'il faut garder à l'esprit lorsque l'on détermine ce que l'on veut qu'il reconnaisse.

5.3 Les améliorations futures

Le but qui a motivé notre projet était d'étudier de nouvelles méthodes pour détecter des attaques, dans un cadre purement expérimental mais qui ne présenterait pas d'obstacle majeur à une utilisation industrielle. Les résultats obtenus dans cette étude ne sont pas suffisamment satisfaisant pour justifier des applications pratiques telles quelles, mais ils ont montré le potentiel incontestable de ses méthodes. Gardons à l'esprit que les deux modèles employés sont des solutions assez générales ne représentant pas de raffinement particulier.

Les raisons pour lesquelles les performances du réseau de neurones sont assez faibles peuvent être nombreuses. Éliminons tout de même la topologie du réseau : nous avons rapidement montré que l'ajout des couches cachées n'augmentait pas la précision du modèle.

On peut commencer une première remise en question avec les paramètres. Ceux-ci ne sont peut-être pas adaptés pour le travail demandé. Il pourrait être judicieux d'étudier, dans un contexte plus général, les logs générés par des attaques informatique, de déterminer beaucoup

de paramètres associés à ces attaques et d'emprunter des techniques à l'analyse de données ou au Big data afin de déterminer lesquels sont réellement porteurs d'information.

De plus on peut imaginer que ce sont les paramètres en eux-mêmes qui ne sont pas adaptés. Perdre le caractère historique d'une trace en la résumant à quelques valeurs est peut-être une erreur et fournir des données sous une autre forme serait plus fiable.

La conséquence directe d'une telle hypothèse est que le modèle en lui-même, c'est-à-dire le perceptron multicouches, ne correspond pas à l'exercice proposé. Les réseaux de neurones couvrent une multitude d'applications allant de la reconnaissance d'images à la génération de résumés. En ce sens, le perceptron multicouches n'est qu'une application parmi d'autres des réseaux de neurones, et des modèles différents exploiteraient mieux les caractéristiques du problème. Les réseaux récurrents, par exemple, pourraient exploiter le caractère historique des logs. Ou bien une machine de Boltzmann réduite, particulièrement performante pour apprendre des distributions de probabilités, aurait des performances supérieures.

Un autre prolongement de notre étude serait de coupler les logs générés par le serveur web avec d'autres logs, par exemple ceux du système d'exploitation. Certaines attaques comme le déni de service ont un impact sur l'hôte du serveur web et des informations précieuses pourraient en être récupérées.

Enfin, il s'agirait de se procurer des données industrielles afin d'éliminer le côté restrictif et artificiel de données générées en laboratoire. En effet, de telles données ne sont pas aussi authentiques que des données réelles et même si elles tendent à prouver la viabilité d'un concept, elles ne préjugent en rien de l'efficacité de ce modèle dans un environnement réel. Notre étude n'est pour le moment pas orientée dans cette direction mais à terme, c'est bien évidemment un objectif à garder en mémoire.

RÉFÉRENCES

[1] 2016. En ligne : `https://httpd.apache.org/docs/2.4/fr/logs.html`

[2] Brodie, M., Ma, S., Rachevsky, L., and Champlin, J. Automated problem determination using call-stack matching. *Journal of Network and Systems Management 13*, 2 (2005), 219–237.

[3] Cook, J. E., and Wolf, A. L. Automating process discovery through event-data analysis. In *Proceedings of the 17th international conference on Software engineering* (225021, 1995), ACM, pp. 73–82.

[4] Dreyfus, G. *Réseaux de neurones : méthodologie et applications*. Eyrolles, 2004.

[5] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[6] Greenwood, R. M. *Using CSP and system dynamics as process engineering tools*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 138–145.

[7] Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks 4*, 2 (1991), 251–257.

[8] Jayathilake, D. Towards structured log analysis. In *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on* (2012), pp. 259–264.

[9] Johns, M. S. Identification Protocol. RFC 1413 (Proposed Standard), Feb. 1993.

[10] Kellner, M. L. Software process modeling support for management planning and control. In *Software Process, 1991. Proceedings. First International Conference on the* (1991), pp. 8–28.

[11] Mariani, L., and Pezze, M. Dynamic detection of cots component incompatiobility. *Ieee Software 24*, 5 (2007), 76–85.

[12] Qiang, F., Jian-Guang, L., Yi, W., and Jiang, L. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on* (2009), pp. 149–158.

[13] Reidemeister, T., Miao, J., and Ward, P. A. S. Mining unstructured log files for recurrent fault diagnosis. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on* (2011), pp. 377–384.

[14] Reidemeister, T., Munawar, M. A., Jiang, M., and Ward, P. A. S. Diagnosis of recurrent faults using log files. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research* (1723031, 2009), IBM Corp., pp. 12–23.

[15] Russo, B., Succi, G., and Pedrycz, W. Mining system logs to learn error predictors : a case study of a telemetry system. *Empirical Software Engineering 20*, 4 (2015), 879–927.

[16] Saeki, M., Kaneko, T., and Sakamoto, M. A method for software process modeling and description using lotos. In *Software Process, 1991. Proceedings. First International Conference on the* (1992), pp. 90–104.

[17] Salzberg, S. L. C4.5 : Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993. *Machine Learning 16*, 3 (1994), 235–240.

[18] Stevanovic, D., Vlajic, N., and An, A. J. Detection of malicious and non-malicious website visitors using unsupervised neural network learning. *Applied Soft Computing 13*, 1 (2013), 698–708.

[19] Vaarandi, R. A data clustering algorithm for mining patterns from event logs. In *IP Operations & Management, 2003. (IPOM 2003). 3rd IEEE Workshop on* (2003), pp. 119–126.

[20] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (1629587, 2009), ACM, pp. 117–132.

ANNEXE A Influence des couches cachées

Afin d'illustrer l'influence des couches cachées, nous allons faire apprendre le logo de Microsoft à un perceptron. Pourquoi le logo de Microsoft ? C'est une image qui n'est soumise à aucun copyright, présentant des zones de couleurs différentes aux frontières géométriques.

Figure A.1 Le logo de Microsoft

Les paramètres d'entrée de nos réseaux de neurones sont les coordonnées x et y de chaque pixel composant l'image et la sortie est le code RGB correspondant à la couleur de ce pixel, incarné par un vecteur de taille 3.

L'apprentissage d'image est un processus lent nécessitant beaucoup de données pour l'ajustement des paramètres. La méthode choisie a donc été l'apprentissage par lots, un lot correspondant à l'ensemble des pixels de l'image. L'image a été fournie 40 000 fois à chaque réseau.

Notons également que le logo de la figure A.1 a une longueur et un largeur de 100 pixels. Cela représente 6 400 données par lot. Étant donnée la complexité grandissante des réseaux, cela peut représenter plusieurs heures de calcul. Afin d'éviter ce problème, l'image a été réduite à un carré de 20 pixels de côté donc 400 données par lot.

La première topologie testée a consisté en une couche cachée de 10 neurones, la suivante en deux couches cachées de 10 neurones, la troisième de deux couches cachées de 50 neurones et la dernière et quatre couches cachées de 50 neurones.

Les résultats ont été représentées dans les figures A.2, A.3, A.4 et A.5. Les temps de calculs pour générer chaque image ont respectivement été d'environ 1 minute et 40 secondes, 4 minutes, 33 minutes et 2h et 48 minutes. Le taux d'erreur était quant à lui d'environ 5% pour les deux premiers modèles et d'environ 2% pour les deux derniers.

On note assez clairement une meilleure capacité du modèle à mieux délimiter les zones de

Figure A.2 [2 - 10 - 3]

Figure A.3 [2 - 10 - 10 - 3]

Figure A.4 [2 - 50 - 50 - 3]

Figure A.5 [2 - 50 - 50 - 50 - 50 - 3]

couleurs lorsque sa complexité augmente, le logo étant assez clairement reconnaissable sur la figure A.5.

Un autre effet illustré par cet exemple est le suraprentissage. Le modèle dont la complexité est la plus importante colle beaucoup mieux aux données que les autres, bien que son erreur ne soit que légèrement plus faible. Cela doit nous forcer à nous interroger sur la complexité idéale de notre modèle et le compromis que nous devons trouver entre précision et généralisation lorsque nous concevons notre réseau.

ANNEXE B Un peu de code

Cette annexe contient des portions du code utilisé durant le projet. Le code étant trop volumineux pour être intégralement retranscrit, certains passages pertinents pouvant répondre à la curiosité du lecteur ont été sélectionnés.

Voici l'algorithme réalisant la rétro-propagation. Il ne fait que les calculs d'erreurs mais ne met pas à jour les coefficients. Dans le cas de l'apprentissage par lots, les erreurs de chaque donnée du lot sont agrégées les unes aux autres et ne sont mises à jour qu'une fois que l'intégralité du lot a été traité.

```
1  public void backTrack()
2  {
3    for (var l = layers.Count - 2; l >= 0; l--) // From the penultimate
         layer to the input :
4    {
5      Neuron[] layer = layers[l]; // Local input layer
6      Synapses synapse = synapses[l];
7      Neuron[] nextLayer = layers[l + 1]; // Local output layer
8      for (int j = 0; j < layer.Length; j++)
9      {
10       double diffSum = 0;
11       for (var i = 0; i < nextLayer.Length - 1; i++) // Calculate the
             error for each connexion
12       {
13         diffSum += synapse.get(j, i) * nextLayer[i].diff;
14         double diff = layer[j].value * nextLayer[i].diff;
15         synapse.addDiff(j, i, diff);
16       }
17       layer[j].diff = sigmoidDeriv(layer[j].input) * diffSum; // Prepare
             for the update
18     }
19   }
20 }
```

Voici la fonction du robot client légitime. Son principe est assez simple : télécharger une page web, récupérer la liste des liens présents sur la page et en suivre un au hasard. Dans le cas où le robot se retrouve sur une page lui demandant de remplir un formulaire, il génère du texte aléatoirement et le soumet. Si des images sont présentes sur la page, il ne faut pas oublier de les télécharger.

```
1  static public void BotClient(int n, List<string> names)
2  {
3    resetLogFile(); // Resets the logfile
4    using (WebClient client = new WebClient())
5    {
6      string userAgent = "Mozilla / 5.0(Windows NT 10.0; WOW64; rv: 46.0)
           Gecko / 20100101 Firefox / 46.0";
7      client.Headers.Add("user-agent", userAgent);
8      string baseURL = "http://192.168.100.131/";
9      string page = client.DownloadString(baseURL + "index.php");
10     HtmlDocument doc = new HtmlDocument();
11     string next = "";
12     int i = 0;
13     Random rand = new Random();
14     while (i < n)
15     {
16       doc.LoadHtml(page);
17       if (next == "livreorsubmit.php") // The case where we submit content
18       {
19         // Select a random name and some random text
20         string name = names[rand.Next(names.Count)];
21         string text = LoremIpsum(10, 20, 2, 4, 1, true);
22         // Add the headers to the request
23         client.Headers.Add("referer", baseURL + next);
24         client.Headers.Add("user-agent", userAgent);
25         // Send the request and go back to the index
26         page = client.DownloadString(baseURL + next + "?name=" + name + "&
               text=" + text);
27         next = "index.php";
28       }
29       else
30       {
31         If images are present, download them
32         HtmlNodeCollection images = doc.DocumentNode.SelectNodes("//img");
33         if (images != null)
34         foreach (HtmlNode node in images)
35         {
36           string imgUrl = node.GetAttributeValue("src", "");
37           client.DownloadFile(baseURL + imgUrl, @"C:\Temp\file.tmp");
38         }
39         // Select a random link and follow it
40         HtmlNodeCollection links = doc.DocumentNode.SelectNodes("//a");
```

```
41          HtmlNode link = links[rand.Next(links.Count)];
42          next = link.GetAttributeValue("href", "pas d'adresse");
43          client.Headers.Add("referer", baseURL + next);
44          client.Headers.Add("user-agent", userAgent);
45          page = client.DownloadString(baseURL + next);
46          if (next == "index.php")
47          i++;
48        }
49      }
50    }
51    saveLogFile(""); // Save the logfile for further analysis
52  }
```

Voici l'implémentation de l'algorithme `kBehavior` présenté à la partie 1. L'implémentation de l'algorithme a été réalisée en utilisant une méthode récursive afin de respecter fidèlement la présentation de l'algorithme par (11).

```
1   static void kBehavior(int k, int kp, Automaton fsa, Trace s)
2   {
3     State q;
4     // Check if the fsa is a new one or an already existing one
5     if (fsa.NumStates == 0)
6     {
7       fsa.traceGenerator(s.prefix(k));
8       q = fsa.States[k];
9       s = s.tail(k);
10    }
11    else
12      q = fsa.State_I;
13
14    while (s.trace.Count != 0)
15    {
16      var split = fsa.splitSeq(s, q);
17      // Get the size of the biggest prefix identified by the automaton
18      // Split the existing trace
19      Trace sPrefix = split.Item2;
20      Trace sTail = split.Item1;
21      // Get to the state reached by the prefix
22      q = split.Item3;
23      // Update the trace
24      s = sTail;
25
26      // Identify a subsequence recognized by the automaton
27      Tuple<Trace, Trace, Trace, State, State> tuple = fsa.identifySub(new
```

```
        Trace(), new Trace(), s, new State(), new State(), k, kp);
28
29      // If no subsequence is found
30      if (tuple == null)
31      {
32        // Check the size of the trace
33        if (s.trace.Count < 2 * k)
34        {
35          Automaton fsap = traceGenerator(s);
36          fsa.merge(q, fsap);
37        }
38        else
39        {
40          Automaton fsap = new Automaton();
41          kBehavior(k, kp, fsap, s);
42          fsa.merge(q, fsap);
43        }
44      }
45      // If a sequence is found
46      else
47      {
48        State qi = tuple.Item4;
49        Trace spre = tuple.Item1;
50        // If the sequence initiates from the initial state
51        if (qi == q)
52        {
53          fsa.createState();
54          State qp = fsa.States[fsa.States.Count - 1];
55          foreach (Transition t in fsa.Transitions.Where(tr => tr.State_f ==
                q))
56          t.State_f = qp;
57          fsa.createTransition(qp, "", q);
58          q = qp;
59        }
60        // Always produce an automaton that generates Spre
61        Automaton fsaPre = new Automaton();
62        if (spre.trace.Count < 2 * k)
63          fsaPre = traceGenerator(spre);
64        else
65          kBehavior(2, 2, fsaPre, spre);
66        // Extend fsa by connecting q to the initial state of fsaPre and the
              final states of fsaPre to qi
67        fsa.merge(q, qi, fsaPre);
68        Trace sprime = tuple.Item2;
```

```
69        q = tuple.Item5;
70        s = tuple.Item3;
71      }
72    }
73  }
```